Testing og feilsøking av program

Testteknikkar for feilsøking i eit SAS-program

Rune Runnestø

Masteroppgåve ved Institutt for informatikk
(60 studiepoeng)

UNIVERSITETET I OSLO

1. mai 2010

Forord

Denne masteroppgåva er utarbeida av Rune Runnestø som avslutning på mastergraden ved Institutt for Informatikk, Universitetet i Oslo.

Masteroppgåva er skriven i samarbeid med Riksarkivet som ekstern institusjon.

Det har difor vore to rettleiarar til oppgåva, Arne Maus frå forskningsgruppa for Objekt-orientering, modellering og språk ved Institutt for Informatikk som intern rettleiar og Lars Nygaard ved Riksarkivet som ekstern rettleiar. Eg vil takka dei begge for bidrag undervegs.

Eg vil også takka dei som har gjeve meg ein barndom og oppvekst som la eit godt grunnlag for å møta verden på ein sjølvstendig og reflektert måte.

Rune Runnestø

Oslo, 01. mai 2010

Samandrag

Masteroppgåva prøver å besvara problemstillingar om val av testteknikkar for testing og feilsøking generelt, samt korleis Riksarkivet sitt SAS-program Arkade best bør testast/feilsøkast.

Testteknikkar har innverknad på kva for feil som blir funne, og i praksis opplever mange testarar at ulike testteknikkar er komplementære til kvarandre (dei finn ulike feil). Men det er også mange andre forhold som har innverknad, og ofte er det uråd å fastslå forklaringsvariablane sin relative innverknad.

Det er mange forhold som er felles for testing og debugging for dei generelle og spesielle problemstillingane. Det indikerer at desse forholda er allmengyldige. Eksempler på slike forhold er utføringen av ein testteknikk, testverktøy, den enkelte feil sin natur, det tekniske testmiljøet, innstillinga til å finna feil, testkompetanse, utviklingsmetodikk og programmerngsstil.

Eg har valt testteknikken utforskande testing som utgangspunkt for å testa Arkade direkte eller til å finna andre testteknikkar som passar til ulike situasjonar. Oppgåva diskuterer fordeler og ulemper med denne testteknikken. Eg brukar testteknikken til å utvikla ein testteknikk for å analysera SAS-loggen. Hensikten er å kvalitetssikra og effektivisera gjennomgangen av SAS-loggen etter kodeeksekvering. Eg brukar også utforskande testing til design av testtilfelle for dynamisk testing av Arkade.

Rapporten konkluderer med at Arkade kan bli lettare testbar ved å forbetra programmeringsstilen. Eit anna forslag er bruk av globale makrovariablar til erstatning for bortkommentering av testkode, slik at aktivering og deaktivering av bortkommentering av testkode i kjeldekoden enkelt kan gjerast ved berre å endra verdien på ein makrovariabel. Rapporten foreslå også at Arkade-applikasjonen bør få implementert større robustheit for feilhåndtering. Kvar makro bør få si eiga programfil, slik at makroane blir lettare å finna når ein treng dei. Applikasjonen kan også gjerast meir testbar ved å aktivera ei rekke systemopsjonar i SAS som gjer at innebygd funksjonalitet for debugging blir brukt.

Dynamisk testing av Arkade ved eksekvering av 23 testtilfelle har avdekka 9 feil. 14 testtilfelle køyrde feilfritt. 1 feil er ikkje funne årsaken til. 8 av feila er kategorisert som dårleg feilhåndtering (2 feil), feil i ein spesifikk del av programmet (3 feil) og for dårleg validering (3 feil) av samsvar mellom inndata-filer (datafiler og metadatafil).

Abstract

The master thesis tries to answer research questions regarding choice of testing techniques to test and debug programs in general, and also how the SAS-program Arkade belonging to the Norwegian National Archives should be tested and debugged.

The choice of testing technique does influence which faults are detected, and many testers experience that different techniqes complement each other. But there are also many other circumstances that matters, and often it is not possible to say how much each factor implies on the total result.

Many of the factors identified are common both for the general and specific research questions studied in this thesis. This indicates that theese factors are of general importance. Examples of such factors are the execution of a testing technique, the testing tool, the nature of the fault, the test environment, the mindsett of the tester, testing experience and test competence, the development method and the programming style

I have selected Exploratory Testing as a starting point for the testing of Arkade. Either by using it directly or by using it to find other techniques which is regarded best suited for the different situations that might appear. The master thesis discusses advantages and disadvantages with this technique. By means of Exploratory Testing I have developed a testing technique for quality assurance of the walkthrough of the SAS log. I have also used Exploratory Testing to design testcases for dynamic testing of Arkade.

The master thesis concludes that Arkade can be more testable by improving the programming style. Another proposal is to use global macrovariables to substitute the comments in the code which masks a lot of code to test the program. Then the testing code can be easily activated and deactivated just by changing the value of the macro variable. The report also suggests to implement better error handling. Every macro should get its very own program file to simplify the process of finding the macro definition when needed. The Arkade application can be made more testable also by activating quite a lot of system options in SAS to make use of built-in features for testing/debugging purpose.

Dynamic testing of Arkade by executing 23 testcases has revealed 9 faults. 14 testcases ran without any faults. For 1 fault I have not found the cause for the failure. 8 of the faults are classified as improper error handling (2 faults), fault in one specific part of the program (3 faults) and too bad validation (3 faults) of possible mismatch between input-files (data files and the metadata file).

Innhaldsliste

Tabelliste

18	Resultat ved testing av testtilfelle 14	Rapporten
19	Resultat ved testing av testtilfelle 15	Rapporten
20	Resultat ved testing av testtilfelle 16	Rapporten
21	Feil og feilmeldingar ved ulike testtilfelle ved testing av Arkade	Rapporten
22	Påviste feil ved dynamisk testing av Arkade	Rapporten
23	Grunnlagsskisse for design av testtilfelle ved dynamisk testing av Arkade	Vedlegg
24	Oversikt over datasett i Arkade der koden overskriv innholdet i datasettet	Vedlegg
25	Oversikt over talet på posisjonsparametre og makrokall for dei 45 makroane i Arkade som har parametre	Vedlegg
26	Datasteg opsjonar relevante for analyse og debuggig av datasteg	Vedlegg
27	Automatiske makrovariablar relevante for debugging av makrospråk	Vedlegg
28	Automatiske variablar relevante for debugging av datasteg	Vedlegg
29	SAS-setningar relevante for debugging av datasteg	Vedlegg
30	Prosedyrer relevante for debugging av SAS-program	Vedlegg
31	Metadatatabellar i SAS relevante for debugging av SAS-program	Vedlegg
32	Oversikt over valte vilkår i 4 eksempler som demonsterer testteknikken for å kvalitetssikra og effektivisera gjennomgangen av SAS-loggen etter eksekvering av programfiler	Vedlegg
33	Eksempel 1 på rapport som viser ekstrakt av viktige meldingar i SAS-loggen	Vedlegg
34	Eksempel 2 på rapport som viser ekstrakt av viktige meldingar i SAS-loggen	Vedlegg

Figurliste

1 Innleiing

1.1 Problemområde

Feil i programvare har volda bry for leverandørar, kundar og kundar av kundane så lenge dataprogram har eksistert. Problema har gjeve seg utslag i kundeflukt samt tapte marknader innanfor det forretningsområdet bedrifta som er avhengig av programvara driv. I tillegg skaper det problem med auka kostnader med å få programvara utvikla på grunn av forseinkingar som typisk oppstår når programvara skal testast.

I den første utgåva i 1974 av boka "The Mythical Man-Month" [1] av Frederick Brooks, skriv forfattaren at "*testing is usually the most mis-scheduled part of programming*". Han refererer til at han sjølv gjennom mange år med stor grad av suksess har estimert testing til å ta ca. 50% av den totale tida for programutvikling fram til levering. Av resten av tida estimerte han 33% til planlegging og 17% til koding. Som eit generelt inntrykk han sit att med frå studiar av andre prosjekt sine estimat, skriv han:

"In examining conventionally scheduled projects, I have found that few allowed one-half of the projected schedule for testing, but that most did indeed spend half of the actual schedule for that purpose. Many of these were on schedule until and except in system testing. Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date. Bad news, late and without warning, is unsettling to customers and to managers".

I dei seinare åra har det vore ein trend mot større bruk av såkalla lettvektsmetodar i systemutviklinga [2]. Dette er arbeidsmåtar som i større grad gjev rom for samtidig utvikling og testing av den same koden. Dette skulle ein tru vil borga for at dei problem som Brooks beskeiv, vil bli mindre.

Men samtidig har det vore ei utvikling mot meir kompleks programvare generelt, og auka konkurranse og fokus på å komma tidleg på marknaden med nye produkt til sluttbrukarar som krev meir i dag enn for nokre år tilbake.

Sidan vi brukar programvare til å debugga [3] programvare, må vi vera bevisste på at feil som oppdagast kan ha si årsak både i den programvara det blir testa med og den programvara som blir testa [4]. Dessutan vil det også kunna introduserast nye feil [5] i programmet som blir testa når oppdaga feil blir forsøkt retta.

Ikkje alle typar program har like store krav til å vera feilfrie. Truleg er ingen program feilfrie. Men program som skal brukast i samanhengar der liv og helse kan avhenga av kvaliteten, har høgare krav til kvalitet enn program der konsekvensane av feil ikkje er så alvorlege.

1.2 Motivasjon

Motivasjonen for å skriva ei masteroppgåve innanfor problemområdet testing av programvare skuldast motivasjon for fagfeltet testing generelt. God testing er ein føresetnad for god kvalitet på programvare, kvalitetssikring og kvalitetsforbetring. Kvalitet er eit positivt lada ord, det gjev assosiasjonar om at "noko fungerer som det skal".

Problemstillingane i masteroppgåva fokuserer på ein avgrensa del av fagfeltet testing. Problemstillingane er motivert ut ifrå at det går ikkje an å testa "alt". Det gjeld å finna dei viktigaste / alvorlegaste feila. Finn for eksempel ulike testteknikkar ulike typar av feil, eller er det grunnlag for å tru at det er dei same feila dei finn?

Eg er medlem i dataforeningens faggruppe for Software testing. Eg har delteke på ein del møter i faggruppa, og tykkjer arbeid med testing har appell.

1.3 Case: SAS-programmet Arkade i Arkivverket

Masteroppgåva er skriven etter avtale mellom Universitet i Oslo og Riksarkivet som ekstern institusjon.

Riksarkivet fekk utvikla SAS-programmet Arkade i tida 1999-2004 av ein ekstern konsulent. Arkade skulle vera eit verktøy for prosessering av tekstfiler og filer i xml-format, og kunna konvertera data mellom ulike filformat. For eksempel konvertera ei xml-fil til ei fil med fast postlengde og felt i faste posisjonar. Eller omvendt.

I 2004 vart utviklinga av programmet stansa av Riksarkivet, fordi det angiveleg skulle vera mange feil i programmet, likevel utan at Riksarkivet dokumenterte overfor leverandøren eksempler på konkrete inndata som førte til feilsituasjonar.

Viss vi ser generelt på fenomenet "mislykka programutviklingsprosjekt", er tilfellet med Arkade neppe unikt eller uvanleg, verken i offentleg eller privat sektor. For eksempel skriv Computerworld [6]:

"Ifølge The Standish Group blir så lite som 29 prosent av it-prosjekter en suksess. Derfor har Computerworlds internasjonale nyhetstjeneste laget en liste over de 14 vanligste feilene, og hvordan man kan unngå dem".

Artikkelen gjev ei oversikt over desse 14 feila.

Bjørnerstedt [7] har ei tilnærming som imøtegår det generelle inntrykket at det er så mange utviklingsprosjekt som blir mislykka. Hans poeng er mellom anna at problemet med studiar som har konkludert med at it-prosjekt har vore mislykka, er at desse studiane har definert feil kva som er mislykka eller vellykka for eit it-prosjekt. I ein oppfølgjande artikkel [8] utdjupar han synspunkta sine:

"Thor-Christian L'orange skriver i CW 25. januar at det er for enkelt av meg å si at et prosjekt kan være vellykket selv om det går over budsjett med 50 prosent. Han feller samme dom over min påstand at man må skille mellom kvaliteten på prosjektstyringen og kvaliteten på prosjektet. Problemet er at han blander sammen spørsmålet om ***hvordan*** *man lykkes i et IT-prosjekt med* ***hva*** *som utgjør et vellykket IT-prosjekt".*

Riksarkivet si rolle ved utviklinga av Arkade var mellom anna å bistå den eksterne konsulenten med testing av den programkoden som vart utvikla. Riksarkivet tok ikkje sjølv direkte del i utviklinga av produktet på kodenivå (programmeringen). Prosessen i forhold til ekstern utviklar er etter det eg erfarar ikkje dokumentert for å gje svar på kva som gjekk gale. Masteroppgåva har ikkje som fokus å forsøka gje svar på kva som gjekk gale med utviklinga av Arkade i SAS. I masteroppgåva er programmet si rolle å vera eit praktisk case for å få svar på spesielle problemstillingar knytt til korleis dette programmet best kan testast, evaluera kor lett testbart programmet er og korleis det eventuelt kan gjerast meir testbart. Eller som Bentley [9] uttrykker det:

"Software testing is a critical element in the software development life cycle and has the potential to save time and money by identifying problems early and to improve customer satisfaction by delivering a more defect-free product. Unfortunately, it is often less formal and rigorous than it should, and a primary reason for that is because the project staff is unfamiliar with software testing methodologies, approaches, and tools. To partially remedy this situation, every SAS professional should be familiar with basic software testing concepts, roles, and terminology. SAS Software provides a comprehensive tool set for building powerful applications. Without adequate testing, however, there is a greater risk that an application will inadequately deliver what was expected by the business users or that the final product will have problems such that users will eventually abandon it out of frustration. In either case, time and money are lost and the credibility and reputation of both the developers and SAS Software is damaged. More formal, rigorous testing will go far to reducing the risk that either of these scenarios occurs".

1.4 Kort om programmeringsspråket SAS

Wikipedia inneheld ein kort historikk om programmeringsspråket SAS og SAS Institute. SAS Institute AS er eit amerikansk programvarefirma med hovudkvarter i Cary, North Carolina, USA. Selskapet blei grunnlagt i 1976 i USA av Anthony Barr, James Goodnight, John Sall and Jane Helwig. SAS Institute er i dag eit av dei største privateigde softwareselskapa i verda [10]:

"SAS har tradisjonelt solgt programvare for statistisk analyse og har vært benyttet mye i forsknings- og universitetsmiljøer. Navnet SAS var opprinnelig et akronym for Statistical Analysis System, men har i mange år vært et varmerke på hele selskapet. Produktene som selges har tradisjonelt vært innen statistisk analyse, men de senere årene har SAS hatt markedsfokus på forretningsanalyse, eller business intelligence".

"Grunnlaget i SAS er et fjerdegenerasjons programmeringsspråk. Over dette programmeringsspråket er det bygget flere forskjellige tykke klienter samt tynne webklienter. SAS benyttes tradisjonelt mye innenfor datavarehusteknologi".

Den delen av SAS som eg kjem i kontakt med gjennom Arkade, er berre ein liten del av systemet [11]:

"SAS (pronounced "sass", originally Statistical Analysis System) is an integrated system of software products".

Kor stor del av det integrerte systemet ein arbeidar med, kjem an på kva for deler som er lisensiert.

Programkoden i Arkade er dokumentert på norsk. Programmet er ein frittståande applikasjon som køyrer på klientmaskiner. Programmet inneheld 75 filer, dei fleste av desse er programfiler. Talet på kodelinjer i desse 75 programfilene er tilsaman mellom 32000 og 33000.

Viktige strukturar i programmeringsspråket SAS er:

- Datasteg
- Datasett
- Makroar
- Prosedyrer
- Funksjonar
- Arrayer

Arkade inneheld 136 makroar, 173 forekomstar av datasteg, 17 forekomstar av arrayer og 156 forekomstar av prosedyrer.

Datasett er ein svært sentral struktur i SAS. Eit datasteg kan resultera i at det vert laga eit datasett, med mindre ein brukar eit såkalla 'data _null_' datasteg. Eit datasett er ein lagringsstruktur for data i SAS [12]:

"Before the warehouse is stocked, before the stats are computed and the reports run, before all the fun things we do with SAS® can be done, the data need to be read into SAS. A simple statement, INPUT, and its close cousins FILENAME and INFILE, do a lot".

"A SAS dataset is a marvelous way to store your data. It is easy to access, easy to query and can contain lots of information about itself. A raw dataset on the other hand is really none of the above; it is a potential mother lode of information waiting to be mined. So, how do you turn these raw data into the gold you want? A SAS data step and the INPUT statement".

For å forstå og kunna debugga SAS-program godt, er det nødvendig med kunnskap om datasteget [13]:

"The DATA step is the most powerful tool in the SAS system. Understanding the internals of DATA step processing, what is happening and why, is crucial in mastering code and output.

This paper focuses on techniques that capitalize on the power of the DATA step and working with (and around) the default actions. By understanding DATA step processing, you can debug your programs and interpret your results with confidence".

Det meste av koden i Arkade er i form av makroar. Ein makro er ein einheit som er utvikla av programmeraren for eit spesielt formål. Dette er såkalla brukardefinerte makroar. I tillegg har SAS funksjonalitet i form av innebygde makroar (desse er ikkje med blant dei 136).

Makroar kan samanliknast med brukardefinerte funksjonar i andre programmeringsspråk. Funksjonar kan også vera innebygde i eit programmeringsspråk. I SAS snakkar ein berre om funksjonar og prosedyrer i tydinga av innebygde funksjonar/prosedyrer.

Howard [14] forklarar skilnaden mellom funksjon og prosedyre slik:

"The fundamental difference between functions and procedures is that a function expects the argument values to supplied across an observation in a SAS data set. Procedures expects one variable value per observation".

Arkade er utvikla i programmeringsspråket SAS. Det er tre "former" for språk i SAS som Arkade inneheld:

- SCL (SAS Component Language)
- SAS-språk
- Makrospråk

Er SAS eit programmeringsspråk som er interpretert eller kompilert? Bakgrunnen for eit slikt spørsmål kan vera ei slik kategorisering av programmeringsspråk som vi finn på Wikipedia [15]. Her kan vi lesa:

"While interpretation and compilation are the two principal means by which programming languages are implemented, these are not fully distinct categories, one of the reasons being that most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" merely mean that the canonical implementation of that language is an interpreter or a compiler; a high level language is basically an abstraction which is (ideally) independent of particular implementations."

Når det gjeld SCL er det eit prekompilert språk [16]:

”The SCL program is automatically compiled and saved when you save the FRAME entry if you set the frame's automaticCompile attribute to Yes.

When you compile a FRAME entry, the FRAME entry's SCL source is compiled, and the compiled code is stored in the FRAME entry. Since the compiled SCL program is stored in the FRAME entry, the SCL entry that contains the source statements does not have to exist when you install and run the application.”

Når det gjeld SAS-spåket med sine datasteg og prosedyrer, blir desse kompilert for kvar gong dei blir eksekvert, dei blir altså ikkje prekompilert [13]:

”There is a distinct compile action and execution for each DATA and PROC step in a SAS program. Each step is compiled, then executed, independently and sequentially.”

Når det gjeld makrospråket, snakkar vi både om åpen og lukka kode. Lukka kode er den delen som er inni ein makro, altså sjølve makroen. Alle andre konstruksjonar i makrospråket er åpen kode. For åpen kode i makrospråket skjer det same som for SAS-spåket med sine datasteg og prosedyrer; koden blir kompilert for kvar gong den blir eksekvert. For makroane er det 3 alternativ, avhengig av kva for makrobibliotek dei blir lagra i [17]:

- For makroar lagra i makrobiblioteket ”%INCLUDE” blir koden kompilert for kvar gong makroen blir kalla (eksekvert).
- For makroar lagra i makrobiblioteket ”Autocall Facility”, blir koden kompilert første gong makroen blir kalla i ein sesjon. For påfølgjande makrokall i same sesjon er koden prekompilert og blir difor ikkje rekompilert, berre eksekvert.
- For makroar lagra i makrobiblioteket ”Compiled Stored Macros” er koden prekompilert og det varer også mellom sesjonar.

1.5 Problemstillingar

Problemstillingane er dels generelle og dels spesielle for det programmet som er case for masteroppgåva.

Generelle problemstillingar

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*
- *I kva grad er ulike testteknikkar komplementære når det gjeld å finna feil?*
- *Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?*

Spesielle problemstillingar

- *Korleis bør Arkivverket sitt SAS-program Arkade testast?*
- *Kor lett testbart er dette programmet?*

- *Korleis kan det eventuelt gjerast meir testbart?*

Problemstillingane er valt for å ta hensyn til at oppgåva skal ha ein generell appell til flest mogeleg, samt at den skal ha ein særleg appell til den institusjonen der masteroppgåva er gjennomført.

1.6 Terminologi

Tilsvarande som problemstillingane er av generell og spesiell karakter, er også terminologien det. I det følgjande gjev eg ei kort orientering om viktige termer som er nyttige å ha eit forhold til for å forstå arbeidet med masteroppgåva.

Generell terminologi for testfaget

Testing av programvare er eit omfattande fagfelt, og som alle fagfelt har det delvis sin eigen terminologi. Vedlegg 1 inneheld ei oversikt over ein del viktige termer for testfaget.

Testing er ein prosess som gjev innsikt i kvalitet og risiko med eit programvareprodukt [18]. Testing omfattar planlegging, utføring og evaluering av aktivitetar for å fastslå om programvareprodukt oppfyller spesifiserte krav og/eller for å demonstrera om dei er brukbare i praksis [19].

Debugging blir definer av Cooper [20] som

> *"the process of locating and correcting errors (bugs) in a computer program".*

Han utdjupar så prosessen ved å dela den opp i fleire steg:

- Observera at det er feil ved programmet
- Isolera kjelda til problemet
- Identifisera opphavet (årsaken) til problemet
- Bestemma seg for ei løysing
- Testa ut den valte løysinga

Cooper poengterer at debugging krev programmeringsferdigheiter og kjennskap til tilgjengelege verktøy og når ein bør bruka verktøya. Han beskriv prosessen som ei blanding av strukturert/systematisk arbeid og instinkt, og at det er ein ferdigheit som kan bli betre ved praktisk trening.

Cooper sin definisjon samsvarar med den Veenendaal og Schaefer brukar [19]:

> *"Prosessen med å finne, analysere og ta bort årsaken til problemer med software (altså feil)".*

Ordet "debugging" finst ikkje i norske ordbøker [21]. Men det finst i norsk talemål. Skal norsk språk beskriva fenomenet utan å bruka den engelske termen, må ein bruka orda "feilsøking", "feilanalyse" og "feilretting". Masteroppgåva har element av alle desse orda i

seg. Viss ein avgrensar seg til applikasjonen Arkade, er det nok mest "feilsøking" som dekker det arbeidet som er gjort. Men oppgåva har også klart element av analyse av feil i kapittel 8. Det er ikkje utført retting av feil i Arkade. Men eg har retta feil i programkode eg sjølv har utvikla for å testa Arkade. Av og til kan det såleis vera ei skjønnsmessig vurdering kvar i oppgåva det er mest dekkande å bruka ordet "debugging" og kvar det er mest dekkande å bruka "feilsøking". Eg ser på ordet "debugging" som så vanleg brukt i fagmiljøet for testing av programvare i Norge at eg har ikkje funne sterke nok argument til å la vera å bruka det heller. For å vera på den sikre sida og ikkje ta munnen for full, har eg brukt ordet "feilsøking" i stadan for "debugging" i tittelen på masteroppgåva.

Problemstillingane i masteroppgåva vedrører eit fagfelt med mange fagtermer. Verken norsk eller engelsk terminologi er eintydig på dette området.

Eit eksempel på det er at det i praksis ofte uråd å skilja termer som ”test”, ”testmetode”, ”testmåte”, ”testtype”, ”testteknikk” , ”deteksjonsteknikk” eller ”feildeteksjonsteknikk” frå kvarandre.

Eit anna eksempel er at det for den norske termen ”feil” ofte blir brukt alternative termer som ”defekt”, ”mangel”, ”avvik” eller ”problem”. Engelske synonym for termen er ”error”, ”fault”, ”failure”, ”defect” , ”bug” og ”incident”. På engelsk blir ”failure” gjerne brukt synonymt med den norske termen ”fiasko”, altså det motsatte av ”suksess” [22], men for praktiske formål går ”fault” og ”failure” også om kvarandre i engelsk språk når konteksten er programvare som ikkje fungerer som den skal. I dokumentet ’Terminologi for test av programvare’ [19] - eit dokument som definerer og samanstiller engelsk og norsk testterminologi - er det ikkje råd å finna nokon skilnad på desse to termene på engelsk heller.

Eit tredje eksempel er ulik bruk av termen ”feiltype”. Eg har erfart at det ikkje finst noko eintydig gruppering i litteraturen. Ei klassifisering er etter i kva utviklingsfase ein feil blir introdusert/produsert. Andre studiar klassifiserer etter dei engelske termene ”omission” (noko som manglar) og ”commision” (noko som er feil). Ein tredje måte er klassifisering etter kor alvorleg feilen er, gjerne karakterisert med fargekodane grøn (ufarleg), gul (uønska, men ikkje kritisk) og raud (kritisk).

Eit fjerde eksempel er termene inspeksjon, gransking og kodegjennomgang. Kodegjennomgang er grei å forstå, jamfør Wikipedia [23]. Inspeksjon er ei meir avgrensa del av gransking (vedlegg 1), men Veenendaal og Schaefer [19] definerar ikkje inspeksjon som avgrensa til andre dokument enn kjeldekoden. Ifølgje Wikipedia [24] kan inspeksjonsobjektet vera "eitkvart som helst arbeid":

"Software inspection, in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well defined process".

Personleg har eg sett føre meg at inspeksjon ikkje vert brukt i tydinga kodegjennomgang. For den empiriske delen har eg differensiert mellom inspeksjon og kodegjennomgang, men for dei referansar eg har (og det er langt flest av dei) kan ein ikkje vita for visst kva referansen har meint viss det ikkje framgår av samanhengen.

Problemstillingane mine søker ikkje å skapa orden eller struktur i dette brokete terminologilandskapet. Problemstillingane er heller ikkje avhengige av at eg sjølv aktivt tek stilling til korleis eg vil definera alle termer i utgangpunktet. Grunnen til at eg bringar terminologi inn i masteroppgåva, er for at lesaren skal vera klar over at det ikkje finnest

nokon allmenngyldige standarder verken nasjonalt eller internasjonalt. Det er snarare slik at termene, som språket generelt, er i kontinuerleg utvikling.

Det er dei aktuelle feila i eit program som er utgangspunktet for klassifisering. Viss metodar ikkje er komplementære, er klassifisering av feil mindre relevant. Ein får også eit problem med å bruka type-termen om feil: Sjølv om ulike studiar brukar type-termen likt, vil ein feil ofte kunna grupperast innanfor fleire typar. Det skaper metodiske utfordringar ved analyse av studiar.

Termen ”feil” i problemstillinga er frå mi side tenkt å forståast som synonym med “feiltype”, jamfør at dei blir forklart som ”feil/feiltype”, jamfør at dei blir forklart som “feil/feiltype” i vedlegg 1. I dette vedlegget skriv eg mellom anna:

“Det er eit poeng at det er ikkje grupperinga i seg sjølv som er avgjerande viktig for debuggingsprosessen, men forståinga av feila sin årsak eller natur. Fellesnemnar for alle grupperingstypar er at kunnskap om SAS er viktig for effektiv debugging. Difor har eg teke med ei forklaring eller eksempler på ein del av feila”.

Spesifikk terminologi rundt Arkade-applikasjonen

Sjølv om testing og debugging i denne oppgåva har fokus sterkast retta mot programkoden, er også studiar av brukarmanual og systemdokumentasjon viktig. I fagforum for testing i regi av Den norske dataforening blir det ofte framheva som viktig at den som testar kode også forstår den forretningsmessige samanhengen applikasjonen skal fungera i. Det utval av uttrykk som er omtala nærmare i vedlegg 2, er uttrykk som eg ser på som kimar til misforståingar. Omtalen av uttrykka er difor ikkje å sjå på som representativ for terminologibruken generelt i systemdokumentasjonen.

Gjennomgangen av uttrykk i vedlegg 2 gjev likevel god forståing for den 3-delte rollemodellen inndata-filene i Arkade har; rollane ‘data’, ‘metadata’ og ‘meta-metadata’.

Dei uttrykka som blir gjennomgått i vedlegg 2 er:

- ADDMML [25] og ADDMML-fil
- makrobibliotek
- metadata-katalog og data dictionary
- datasett
- databasar

1.7 Undersøkingsmetodar

I utgangspunktet var masteroppgåva tenkt gjennomført med eksperiment som overordna design. Men i løpet av arbeidet med oppgåva stod det klårt for meg at føresetnadane *ikkje* ligg til rette for at det innanfor dette masterarbeidet kan utførast samanliknande *eksperiment* som vil kunna ha nokon som helst verdi når det gjeld ekstern validitet. For å bruka eit analogt bilde: “Ei svale gjer ingen sommar”. Innsamling av materiale frå praktisk testarbeid med

tanke på statistisk bearbeiding vil neppe la seg gjennomføra. Men det tok ein god del tid å innsjå det. I stadan har eg valt å la eksperiment vera representert *indirekte* i masteroppgåva ved at eg presenterer relevante resultat frå relevante artiklar.

Dei metodane som er brukt for for å finna svar på problemstillingane i denne masteroppgåva er

- teori både i form av litteraturstudiar og deltaking på Software 2010
- strukturerte intervju
- case-studium

Gjennom arbeidet med problemstillingane søker eg å bryta teori mot teori, teori mot andre sin erfaring, teori mot eigne erfaringar, og for såvidt også andre sin erfaring mot eigen erfaring. Det heile byggjer på *kvalitative* vurderingar (som veldig mykje gjer innanfor Software Engineering) for å finna ut om noko peikar seg ut som generaliserbart.

Vekslinga mellom teori og empiri er metodetriangulering i praksis. Røykenes [26] forklarar metodetriangulering som

"...at bestemte fenomen studeres fra ulike synsvinkler og synspunkt, og problemstillingen belyses ved hjelp av forskjellige metoder og data (Grønmo, 2004). Denzin definerer triangulering som "en kombination af metodologier, der bruges ved undersøkelse af samme fænomen" (Denzin i Kruuse, 2001 s.47). Selve ordet triangulering stammer fra landmåling og navigasjon, og er en prosess hvor en benytter to punkt for å finne den ukjente avstanden til et tredje punkt. Enkelte forfattere (blant annet Williamson, 2005) mener triangulering innen forskning bør sees på som en metafor for denne prosessen".

1.8 Strukturen i masteroppgåva

I dette punktet gjev eg ein kort introduksjon av kva som er tema i dei ulike kapitla vidare i masteroppgåva.

Kapittel 2

Teoretisk gjennomgang av 5 studiar som fokuserer på dei generelle problemstillingane.

Kapittel 3

Ustrukturerte intervju med 3 testarar som fokuserer på dei generelle problemstillingane.

Kapittel 4

Formidling av erfaringar frå Software 2010 som er relevante for dei generelle problemstillingane.

Kapittel 5

Diskusjon og samanstilling av resultat frå kapittel 2, 3 og 4 som fokuserer på dei generelle problemstillingane.

Kapittel 6

Teoretisk gjennomgang av ulike skular eller historiske tilnærmingar til testing av programvare, for å kasta lys over dei spesielle problemstillingane.

Kapittel 7

Teoretisk gjennomgang av testteknikken utforskande testing (vedlegg 1, sjå også [27]). Kva den er, når det passar å bruka den og ulike synspunkt på den. Alt saman for å søka nærmare svar på dei spesielle problemstillingane.

Kapittel 8

Teoretisk og praktisk kapittel som forsøker å gje svar på dei spesielle problemstillingane. Den empiriske delen av masteroppgåva er samla i dette relativt store kapittelet.

Kapittel 9

Diskusjon om nytten av testteknikken utforskande testing. Diskusjon av dei spesielle problemstillingane, og desse sett i samanheng til dei generelle problemstillingane.

Kapittel 10

Konklusjon med samanfatning av funn, kritikk av eige arbeid, refleksjon over kva masteroppgåva kan bidra med og kva som peikar vidare framover.

Vedlegg

Det er 15 vedlegg til masteroppgåva. Vedlegga er i den rekkefølgja dei blir referert i rapporten. Sidan vedlegga tilsaman er på 175 sider, er det valt å trekka dei ut frå den trykte utgåva av masterrapporten. Dei ligg vedlagt som ei pdf-fil på ei CD-plate. Sidenummereringa for den elektroniske fila med vedlegg startar på side 140, som er same sidetal som vedlegga ville starta på om dei hadde komme etter rapporten.

2 Gjennomgang av artiklar om testmetodar

Dette kapittelet går nærmare inn på fagfeltet testing, og forsøker å gje svar på dei generelle problemstillingane med referanse til relevante artiklar. Dei generelle problemstillingane er:

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*
- *I kva for grad finn ulike testteknikkar ulike feil?*
- *Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?*

Eg søkte etter relevante artiklar i databasane ISI Web of Knowledge, ACM Digital Library, Bibsys Ask og IEEE Xplore. Søkemotoren Google har vore flittig brukt.

I litteratursøk for å finna relevante artiklar har eg brukt forskjellege søkeord, både enkeltvis og i kombinasjon. Dei enkeltvise søkeorda er ”Error detection”, “Error detection in software”, “Testing method”, “Error detection method”, “Software testing”, “Comparing testing methods”, “Komplementære testteknikker”, “Complementary testing techniques”, “Complementary testing”, “Comparing debugging techniques”, “Complementary debugging techniques” og “Complementary error testing techniques”.

Eg har plukka ut fem artiklar frå studiar som delvis er relevante for problemstillingane mine. Ei avgrensing i masteroppgåva er at eg ikkje går detaljert inn i kvar artikkel for å evaluera evidens for påstandar eller konklusjonar som blir framsatt. Fordi det vil sprenga rammene og dermed balansen og fokuset for det eg ønskjer masteroppgåva skal bidra med. Eg nøyer meg difor med i all hovudsak å trekka ut poeng av det forfattarane sjølve meinar dei kan konkludera med, og som samtidig er relevant for problemstillingane mine.

2.1 Artikkel 1

Tittel på artikkelen er: “Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect?”.

Artikkelen av Juristo og Vegas [28] omtalar 2 eksperiment som prøver å gje svar på tre testteknikkar sin evne til å oppdaga feil: Funksjonell testing (vedlegg 1) strukturell testing (vedlegg 1) og kodegjennomgang [23]:

“We presented two successive experiments that aim to clarify the fault detection capability of three code evaluation techniques: two dynamic analysis (functional and structural) and one static analysis (code reading by stepwise abstraction) techniques.”

Det første eksperimentet kunne tyda på at funksjonell testing kom best ut:

“The design of the first experiment included four programs, four fault types per program (although they contain two faults of three of the types, adding up to a total of nine faults per program) and each subject applied a single technique. From Experiment I, we

found that the cosmetic faults showed up worst (irrespective of the technique and the program) and that code review was not affected by the fault type (they were all detected equally). This contrasts with Basili and Selby's findings, of review detecting better certain types of faults, and Kamsties and Loot's finding of no difference between fault type. As regards the actual techniques, the functional technique came out better than the structural technique for faults of omission (the same finding as Basily and Selby) and the structural technique behaved equally or better than the functional technique for faults of commission (Basili and Selby did not find this, but all behaved equally), although, on the whole, the functional technique tended to behave better (as in one of the experiments of Basili). However, we did not manage to discern a clear behaviour pattern as regards what fault types each of the two techniques detects better (contrasting with Basili and Selby's findings)".

Men så kom ein til at det var metodiske manglar i eksperiment 1, som ein prøvde å ta hensyn til med eksperiment 2:

"Some of the findings of Experiment I, however, led to the preparation Experiment II. The fact that not all the faults were replicated in Experiment I led to the creation of two versions of each program in the second experiment. As replicated faults behaved differently (which was strange), we thought that perhaps it could be the failure that caused this variation. Therefore, we examined failure visibility during Experiment II. For this purpose, we generated test cases that detected all the faults and asked the subjects to use these test cases instead of running their own. As the faults did not appear to be having an impact on the static technique, we decided to investigate fault visibility in the static technique in Experiment II. Finally, we also suspected that the subjects might have influenced fault detection in Experiment I, which led us to have all the subjects apply all the techniques rather than just one as in Experiment II".

Og resultatet vart at eksperiment 2 tilbakeviste resultatet i eksperiment 1:

"Experiment II was run on the basis of these premises. The findings from the experiment corroborated some of the suspicions we had had during the first experiment. Firstly, as regards the possible impact of the fault and the technique, we found that the functional and structural techniques behaved identically (this contrasts with Basili and Selby's findings of functional technique behaving better). This refutes the finding from Experiment I that the functional technique behaved better than the structural technique for some fault types. This is because there was a hidden effect in Experiment I, namely, failure type. This was influencing failure detection".

Det resultat ein var ståande tilbake med, var at testerfaring og kompetanse til å sjå feil er minst like viktig som testteknikken i seg sjølv:

"As regards failure visibility, this does indeed have an influence. Hence, we have been able to establish a failure taxonomy, where error messages that do not appear and incorrect results are the most visible. So, the results observed in Experiment I were due to failure visibility not to the power of the techniques. It is not clear, however, whether perhaps the functional technique, owing to its modus operandi, tends to make subjects more sensitive to

the detection of certain faults. From this we can deduce that it is not only important to teach subjects the testing techniques, but it would also make sense to teach them to see the failures".

Ei erfaring ein meinte å kunna trekka frå eksperimentet var at programmet er viktig for om ein feil blir oppdaga:

"Additionally, a new thing we found was that the program/technique combination has an impact on the number of faults detected (as Wood et al. already found), although the functional and structural techniques again behaved equally in all cases for the same program. This means that technique effectiveness (irrespective of whether it is the structural or functional technique) is affected by the program. In other words, whatever the technique (structural or function) we apply, it will always be less effective for a given program type than for another".

Eit anna forhold gjorde at erfaring vart vurdert som viktig for å finna feil:

"Another interesting result was that the position of a fault has no influence on the number of people who see the fault using the reading technique. This suggests that we should look for other factors that may have an impact (perhaps experience, which has been addressed in earlier experiments). Although neither of these experiments has taken this into account, aspects like subject experience have been investigated in earlier experiments examining code review effectiveness".

Artikkelen konkluderer med at det er den enkelte feil heller enn den type den måtte vera klassifisert under, som avgjer kor lett den er å oppdaga:

"Finally, another unexpected finding was that the actual version influenced the number of subjects who were able to generate a test case that detected a fault. We have interpreted this as it being the actual fault, that is, the particular instance and not the type of fault, rather than the program type or form that determines whether more or fewer faults are detected. This leaves the field open for further research".

Forfattarane har brukt den same inndelinga av feil som Basili og Selby [29]. Forfattarane er klar over begrensningen her og forklarar effekten på ambisjonsnivået greit nok: *"...the study is not intended to be exhaustive, but rather to test whether the hypothesis is true*".

Eit svakt punkt i studien med tanke på ekstern validitet (vel og merka når industi-kontekst er referanseramma for ekstern validitet), er at det er små program med berre 400 kodelinjer (inkludert blanke linjer) i kvart av dei 4 programma i eksperiment 1. Artikkelen opplyser at dei injiserte feila er "representative" for feil i industri-kontekst. Korleis ein kan vita det, seiest det ingenting om. Ei anna sak er at måten ein feil oppstår på, kan tenkjast å influera på kor lett feilen blir å gjenfinna. Det er forskjell på det å skapa ein feil kunstig, og det å skapa den utan vitande og vilje. I det første tilfellet kan vi aldri gardera oss mot at den som injiserer feilen, kan ha eit – bevisst eller ubevisst – ønske om at feilen skal finnast igjen. Det kan ikkje utelatast at det kan påverka det faktiske resultatet.

2.2 Artikkel 2

Tittel på artikkelen er: "An empirical evaluation of defect detection techniques".

Artikkelen av Roper, Wood og Miller [30] er ein empirisk studie (eksperiment) frå 1997 som samanliknar tre teknikkar sin evne til å oppdaga feil: Inspeksjon (vedlegg 1), funksjonell testing og strukturell testing. Teknikkane blir utført på 3 program på ca. 200 kodelinjer kvar. Hovudkonklusjonen er at kombinasjon av fleire teknikkar aukar talet på feil ein finn:

"The results reported here appear to show that the different techniques have different strengths and weaknesses in terms of the faults that they help to uncover. Their absolute effectiveness as well as their relative effectiveness depends on the nature of the programs and more specifically on the nature of the faults in those programs. This is supported by the significant body of related work which shows no consistent pattern in terms of absolute or relative effectiveness. Rather, as the programs and faults vary so do the results".

"Secondly, and building on the finding that these different techniques are to some extent orthogonal, it has been shown that they are much more effective when used in combination as opposed to being considered alternatives. Again, it is argued that this is consistent with earlier findings that the techniques vary in their effectiveness depending on the nature of faults".

Forfattarane meinar også at studien langt på veg støttar synet om at ulike testteknikkar er komplementære:

"To a large extent these findings are consistent with software engineering folklore. Standard software engineering texts teach that the basic techniques are complementary. It does appear that there is a body of empirical evidence developing that supports those beliefs".

"The empirical study reported in this paper has now been carried out in various forms at five different sites over 20 years. Evidence from those studies was used to explain and substantiate the current research findings".

2.3 Artikkel 3

Tittel på artikkelen er: "An Experimental Evaluation of Inspection and Testing for Detection of Desing Faults".

Artikkelen til Andersson, Thelin, Runeson og Dzamashvili samanliknar testteknikkane inspeksjon og testing [31]. Forsøkspersonane er 51 studentar som går fjerde året i ei masterutdannig i Software Engineering. Studentane vart fordelt til grupper tilfeldig. Kun 1 av studentane fullførte ikkje eksperimentet:

"This paper reports an experiment on two fault detection techniques, usage-based reading (UBR) and usage-based testing (UBT), conducted with 51 Master's students in

December 2002. The results show that UBR is significantly more effective and efficient than UBT. The results are slightly different for the two experiment groups but the overall results are in favour of UBR. The results indicate that it takes longer to learn the UBT. Comparing the techniques independently of program versions, UBR is significantly better than UBT both in terms of efficiency as effectiveness. The experiment also investigated whether different faults were found by the two techniques. One group shows statistically significant differences while the other does not. If the techniques find different faults, it implies that they should be used as complements".

"Given that the results can be replicated and generalized, we conclude that this study provides evidence that usage-based reading for design inspections are more effective and efficient than usage-based functional testing. This holds for the fault detection as such, which is the focus of this paper. When taking the rework costs into account, the potential gains are larger with the inspection technique since it is possible to apply earlier in the development cycle".

Sjølv om artikkelen finn den eine metoden meir effektiv enn den andre, har den ikkje noko svar på korleis metodane bør kombinerast for å få best resultat:

"Further work should include further experimentation to replicate the results. One focus could be on which types of faults are found by the different techniques. This would give better answers to whether the two techniques find different types of faults or not, and could facilitate the decisions of how to combine inspection and testing, and answer the question about how to reduce these effort consuming activities. Furthermore, conducting an experiment with the testing treatment in a dynamic test environment would reduce the threat to construct validity of having the paper-based approach".

2.4 Artikkel 4

Tittel på artikkelen er: "Studying the effects of code inspection and structural testing on software quality".

Laitenberger sin artikkel [32] beskriv eit forsøk der 20 vidarekomne studentar ved Computer Science Department ved eit universitet i Tyskland har gjort eit kontrollert forsøk og testa eit program i språket C med kodegjennomgang og strukturell testing. 13 feil er injisert i programmet. Han konkluderer med at strukturell testing og kodegjennomgang i liten eller ingen grad er komplementære når det gjeld å finna feil i program:

"This paper investigated the effects of combining software inspection and structural testing on software quality. In contrast to the claim, that each technique has its own merits and, therefore, should be applied in combination, the results of a controlled experiment with students as subjects showed little empirical evidence that this claim is justified.

"In the experiment, software inspection and structural testing did not complement each other well, that is, defects subjects missed in inspection were often not detected by subjects applying structural testing".

Laitenberger poengterer at dette forsøket viser eit resultat når to teknikkar blir samanlikna, men at ein av det ikkje kan konkludera med at ulike testteknikkar generelt ikkje komplementerer kvarandre:

"This result is supported by our finding that inspection outperforms structural testing with respect to effectiveness and that both defect detection techniques do not focus on the detection of defects of different defect classes. However, whether other testing approaches, such as boundary value analysis, are really more complementary to inspection than structural testing is in need of further experimentation".

Eg synest følgjande vurdering av forsøket si vekt høyrest truverdig ut:

"We consider this experiment as only one step in the determination of the optimal mix of defect detection techniques. Additional research as well as replication of this experiment are required to develop a better understanding of defect detection techniques and make further progress in this direction".

2.5 Artikkel 5

Tittel på artikkelen er: "What Do We Know about Defect Detection Methods?"[33].

Denne rapporten skiljer seg ifrå dei andre ved at den er ein meta-analyse; ein analyse som analyserer og evaluerer resultat frå fleire andre studiar [34]:

"A systematic literature review is a means of evaluating and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest. Systematic reviews aim to present a fair evaluation of a research topic by using a trustworthy, rigorous, and auditable methodology".

Meta-analysen omfattar 12 studiar, herav 10 eksperiment og 2 case studiar.

Artikkelen er litt meir popularisert i forma enn dei 4 foregåande, men med 22 referansar verkar inntrykket av fagleg belegg godt nok.

Den mest visuelt iaugefallande konklusjonen (utheva med feit grøn skrift øverst på første side i artikkelen) lyder:

"A survey of defect detection studies comparing inspection and testing techniques yields practical recommendations: Use inspections for requirements and design defects, and use testing for code".

Forfattarane synest ha like liberal holdning til bruken av termer som eg har gjeve uttrykk for tidlegare i denne oppgåva:

"The term defect always relates to one or more underlying faults in an artifact such as code. In the context of this article, defects map to single faults. Thus, we use the terms defect and fault interchangeably, as have many of the authors whose work we refer to".

Artikkelen reflekterer at den forstår dilemmaet med ulike termer, og gjer greie for sitt eige val for klassifisering av testteknikkane:

"Which techniques are you using for inspection and testing respectively? Because there are many techniques for each, we refer to inspection and testing as families of verification techniques. For testing, we distinguish between structural (white box) and functional (black box) testing".

Artikkelen refererer til mange andre liknande studiar, over ein 25-årsperiode, og inntrykket som formidlast er at det fortsatt er mykje ein ikkje veit:

"Many empirical studies have investigated defect detection techniques, inspections, and testing in isolation. Aybüke Aurum, Håkan Petersson, and Claes Wohlin summarize 25 years of empirical research on software inspections, including more than 30 studies that investigated different reading techniques, team sizes, meeting gains, and so on. Similarly, Natalia Juristo, Ana Moreno, and Sira Vegas summarize 25 years of empirical research on software testing based on more than 20 studies. They compare testing within and across so-called "families" of techniques. Despite the large number of studies, they conclude that our collective knowledge of testing techniques is limited".

Det synest vera liten tvil at det er lønnsomt å få luka ut feil allereide i kravspesifikasjonen:

"The choice between requirements inspection and system testing is quite obvious and needs no experiments: spending effort up front to establish a good set of requirements is more efficient than developing a system on the basis of incorrect requirements and then reworking it".

I eit forsøk det blir referert til, viste inspeksjon seg overlegen systemtesting (vedlegg 1, sjå også [35]) for å oppdaga feil som var introdusert i designfasen:

"Carina Andersson and her colleagues addressed this issue in one experiment. They observed defect detection in a design specification and in a log from function test execution; they used experimental groups that varied greatly. Inspections were significantly more effective and efficient than testing. The study's participants found more than half of the defects (53.5 percent) during inspection and fewer (41.8 percent) during testing (see table 3). Efficiency was five defects per hour for inspection and fewer than three per hour for testing.".

I eit par andre forsøk det blir referert til, blir desse resultata støtta:

"Berling and Thelin confirmed these results in their industrial case study, including five incremental project iterations.3 Inspections detected on average 0.68 defects per hour, while testing detected 0.10 defects per hour (see table 3). For the fraction of faults that they estimated to propagate into code, the rate was 0.13 defects per hour. This case study also

reports slightly higher effectiveness for inspections, but the differences are small. Reidar Conradi, Amarjit Singh Marjara, and Børge Skåtevik had similar results in their case study: they reported 0.82 defects per hour in design inspection and only 0.013 defects per hour in function testing. So, the empirical data support design inspections as a more efficient means for detecting design defects".

Når det gjeld feil introdusert under programmeringa, er rapporten noko uklar. På ein stad i artikkelen blir det sagt at det er ingen klar preferanse med hensyn på testteknikk:

"Most of the experiments investigated code defects. However, they revealed no clear answer as to whether code inspection or testing—functional or structural—is preferable".

"The data doesn't support a scientific conclusion as to which technique is superior, but from a practical perspective it seems that testing is more effective than code inspections".

Men på ein annan stad i artikkelen er det preferanse for funksjonell og strukturell testing framfor inspeksjon:

"For code, functional or structural testing is ranked more effective or efficient than inspection in most studies. Some studies conclude that testing and inspection find different kinds of defects, so they're complementary. Results differ when studying fault isolation and not just defect detection".

Artikkelen gjev uttrykk for at samanlikning av feil på tvers av ulike forsøk er vanskeleg, fordi klassifisering og bestemming av feil er mykje avhengig av skjønn:

"Comparing the studies is difficult. First, only five of the studies report defects by the same type scheme and can be used to investigate whether the differences depend on the fault type, not only on the technique. Second, the frequencies of the different defect types vary widely among the remaining studies".

"Fourth, classification schemes involve subjective judgment that can confuse classification results".

Det er nyttig å vera klar over at det er normalt at programvare har feil i seg:

"Absolute levels of effectiveness of defect detection techniques are remarkably low. In all but one experimental study, the subjects found only 25 to 50 percent of the defects on average during inspection, and slightly more during testing (30 to 60 percent). This means that on average, more than half the defects remain!"

"The Berling and Thelin case study reported 86.5 percent effectiveness for inspections and 80 percent effectiveness for testing. However, these are based on an estimated number of defects that the technique could possibly find, not on the total number of defects in the documents".

Ein praktisk implikasjon av så relativt liten del av alle feil blir oppdaga før eit system blir teke i bruk, blir å leggja større vekt på feilfinning blant sluttbrukarar:

"The practical implication of the primary defect detection methods' low effectiveness and efficiency values is that secondary detection methods might play a larger role than the surveyed empirical studies concluded".

Ein av dei generelle problemstillingane er kva for andre faktorar enn testmetoden i seg sjølv som spelar ei rolle for oppdaging av feil. Artikkelen nemner ei rekke forhold, utan å gå nærmare inn på nokon av dei:

"The choice of defect detection method depends on factors such as the artifacts, the types of defects they contain, who's doing the detection, how it's done, for what purpose, and in which activities. Factors also include which criteria govern the evaluation. These factors show that many variations must be taken into account. When you search the evidence for the the pros and cons of using some defect detection method, you must choose specific levels of these factors to guide the appraisal of empirical evidence.use inspections for requirements and design defects, and use testing for code".

3 Praktisk erfaring frå testarar

Det har vore eit bevisst mål med masteroppgåva at den skal ha eit praktisk tilsnitt samtidig som den skal forsøka trekka ut erfaringar frå gode og relevante studiar.

I dette kapittel presenterast resultat av tilbakemeldingar frå tre testarar. Testarane er anonymisert og referert til som testar A, B og C. Alle tre arbeidar med testing i programvareindustrien til dagleg.

Testar A har arbeidd med testing i 8 år, og er avdelingsleiar i testsenteret der han no arbeidar. Han har vore styremedlem i Den Norske Dataforening si faggruppe for software testing. I testarbeid har han hatt mange ulike rollar, som for eksempel systemansvarleg, versjonsansvarleg, testleiar, testdesigner, testar, testmiljøansvarleg og testdataansvarleg.

Testar B har arbeidd med testing dei siste 6 år. Før det var han utviklar og prosjektleiar i 3 år, med fokus på testing, spesielt automatisert testing (vedlegg 1, sjå også [36]). Han har mellom anna arbeidd med teststrategi, testplanar, utarbeiding av testtilfelle (vedlegg 1, eller sjå [37]) for manuelle testar, testtilfelle for ikkje-funksjonelle testar (vedlegg 1, eller sjå [38]) og ytingstesting (vedlegg 1, eller sjå [39]).

Testar C har mange års erfaring med testing frå rollen som utviklar. Han har arbeidd med einheitstestar [40], lågnivå ingtegrasjonstestar, funksjonelle testar og systemnivå integrasjonstestar [41].

Som kapittel 2 er også dette kapittelet tenkt å gje svar på dei generelle problemstillingane:

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*
- *I kva for grad finn ulike testteknikkar ulike feil?*
- *Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?*

Frå alle tre testarane foreligg skriftleg tilbakemelding på epost. For A og B har eg i tillegg gjennomført samtale over telefon. Samtalane var avtalt på førehand og vart utført som ustrukturerte intervju. Det var lause rammer for samtalane, og testarane vart gjort klar over kvifor det var eit bevisst val frå mi side. Eg forsøkte å leggja vinn på å ikkje "styra" samtalane i særleg grad, for på den måten å leggja best til rette for at testarane skulle få komma fram med deira synspunkt og tilnærming til problemstillingane utan å bli avgrensa av ein regi som frå mi side ubevisst kunne ha eit gjeve resultat som formål.

3.1 I kva for grad finn ulike testmetodar ulike feil?

Erfaringar frå testar B

B har erfart at når fleire personar køyrer dei same testtilfella, finn dei stort sett dei same feila. Men to testteknikkar utmerkar seg ved å finna andre feil enn dei spesifiserte testtilfella gjer, nemleg utforskande testing og destruktiv testing (vedlegg 1, eller sjå [42]).

B synest det er vanskeleg å vera sikker på om inspeksjon og dynamisk testing (vedlegg 1) er komplementære. B prioriterer inspeksjon berre på utvalte deler av programmet. For andre deler blir det ein mindre grundig gjennomgang. B har erfart at inspeksjonar eignar seg godt til å oppdaga logiske feil. I tillegg gjev det muligheit til betre vedlikehold (bruk av felles kodestandarder) og gjer at fleire blir kjent med koden i programmet.

Ifølgje B arbeidar ein utifrå ei overtyding om at ulike testteknikkar finn ulike feil. Men det er vanskeleg/umogeleg å få statistikk frå testarbeidet som styrkjer eller svekkar denne trua. Som indikasjon på om målet med testteknikken er oppnådd, brukar ein å måla dekningsgraden av funksjonalitet. Dette vil sei noko om kor grundig testen er, men er eigentleg ikkje evidens for om testteknikken eller miksen av testteknikkar finn andre eller dei same feil som ein annan kombinasjon av teknikkar ville funne. Industrien har verken tid eller kapasitet til å få svar på kva for resultat dei alternativa teknikkane som ein ikkje veljer, ville gjeve.

Sjølv om formålet med testteknikkar er å finna ulike typar feil, har B erfart mange eksempler på at testteknikkar finn andre typar feil enn dei ein forventar eller hadde laga testen for. Slike uventa feil kan både visa eller ikkje visa at ein teknikk er komplementær til ein annan teknikk. Det kjem an på kva for andre teknikkar ein brukar. B har erfart at uventa feil ofte blir funne ved bruk av funksjonelle testar. Dette blir forklart med at dei funksjonelle testane skal dekka mest mogeleg av applikasjonen sin breidde og djubde. Funksjonelle testar blir utført etter ein har køyrt dei automatiserte einheitstestane. B har erfart at dette ofte er det tidlegaste i utviklingsforløpet at åpenbare feil kan bli oppdaga.

B har også erfart at testar kan avdekka uventa årsakssamanhenger. For eksempel fann han feil i eksternt nettverk ved ytingstesting. Årsaken til feilen var ein brannmur 30 mil frå testmiljøet (mellom to system). Bentley [9] definerer testmiljø som “The technical environment, data, work area, and interfaces used in testing”.

Erfaringer frå testar C

C har erfart at einheitstestar finn feil som utviklaren sjølv kjem på. Dette er ofte fleire feil enn ein skulle tru, men på langt nær nok til å rekna programvara som veltesta. Einheitstestar kan imidlertid vera ein kostnadseffektiv måte å hindra regresjon av feil som oppdagast i andre testar.

Ifølgje C si erfaring kan rett bruk av lågnivå integrasjonstestar finna ein vesentleg mengde feil som einheitstestar ikkje dekkar. Ofte er det denne type testar ein kan bruka for å finna feil i bruk av servlet-filter, databasemapping, html-skjermbilder etc. Feil brukt kan lågnivå integrasjonstestar bruka unødvendig lang tid på å testa ting som elles hadde vore lurt å testa med einheitstestar.

C har meddelt sine erfaringar med ulike testmetodar for ulike feil, der han har kategorisert feil etter årsaksgrupper:

”Feil i implementasjonen av utviklerens forståelse oppdages av enhetstester. Det vil typisk oppdages av andre nivåer med tester, men ikke like kostnadseffektivt.

Feil i utviklerens forståelse av kravene kan oppdages av funksjonelle tester eller brukertester. De er mest effektive å oppdage i funksjonelle tester.

Feil i kravbeskrivelse vil oppdages kun i brukertest.

Feil i logikken i skjermbilder (eks JSP) vil oppdages i lavnivå integrasjonstester, funksjonelle tester eller brukertester.

Feil i bruk av infrastruktur vil oppdages i integrasjonstester, enten på lavnivå eller systemnivå. Det er mye billigere å finne så mange som mulig av disse feilene på lavnivå tester.

Ytelses og stabilitetsproblemer oppdages som regel i systemnivå integrasjonstester."

C meinar at systemnivå integrasjonstestar er eineste måten å finne feil i produksjonsaktig konfigurasjon, nettverksproblemer etc. Han har brukt denne testmetoden til å testa robustheit, skalerbarheit og toleranse for uventa data, og har erfart at ved å bruka strategiar for å generera uvanleg belastning og dataverdiar kan systemnivå integrasjonstest finna feil som tidlegare testar ikkje fann. Verktøy som JMeter og Grinder hjelper til å utføra slike testar, men mange team lagar også eigne script for dette.

3.2 Er feiltypane som identifiserast avhengige av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?

Generelt er svaret på denne problemstillingen "både-og" frå testarane. Først presenterast erfaringar som tilseier at testmetoden i seg sjølv er viktig, deretter andre forhold som dei også meinar spelar inn.

Testmetoden i seg sjølv

A har erfart at for mange testteknikkar ligg det i deira natur at dei skal finna ulike feil. Og dei gjer det erfaringsmessig. Eit eksempel på dette er negativ testing versus positiv testing (vedlegg 1). Når testar har ulikt fokus, vil dei finna ulike feil. Negativ (også kalla destruktiv) testing vil fokusera på å skriva tekst i felt for tallverdiar og omvendt eller bevisst overskrida grenseverdiar (vedlegg 1). Dei type feil dette fører til, vil ved retting gje auka driftsstabilitet. Postitiv testing vil testa om systemet fungerer som det skal med forventa inndata. Eit anna eksempel er ytingstesting versus stresstesting (vedlegg 1 eller sjå [43]). Ytingstesting testar belastning over tid (antal brukarar pålogga), mens stresstesting typisk testar overbelasting i form av transaksjonar per tidseining. Eit tredje eksempel er funksjonelle testar versus livsløpstestar. Funksjonell testing testar det som er endra i eit aktuelt system. Livsløpstest involverer gjerne fleire system, eksempelvis ein applikasjon for import av køyretøy, som skal henta data frå ulike fagsystem. Eit fjerde eksempel er manuell versus automatisert testing. Ved manuell testing (vedlegg 1) har den som utfører testinga kontroll på testinga, mens ved automatisert testing køyrer ein testscript som ofte er utvikla av andre personar enn den som brukar dei i testinga.

I praksis stillest ofte knapp tid til disposisjon for testing, noko som ofte fører til at valet fell på automatiserte framfor manuelle testar. Som eit femte eksempel viser A til ei diplomoppgåve [44] han var med og skreiv hos eit firma som gjennomførte og samanlikna dei

to teknikkane utforskande testing og ”tradisjonell” testing (testing med både kvitboks testing [45] og svartboks testing [46]). Programvara som var objekt for studien var i vedlikeholdsfasen. Resultatet frå denne studien var at ein fann 11 feil med tradisjonell testing og 13 feil med utforskande testing, og at alle feila var ulike. Altså at desse to testmetodane var komplementære i stor grad. Dei to testgruppene var begge rekruttert frå utviklingsavdelinga internt i firmaet.

B støttar desse erfaringane med sine eigne ved å visa til at i praksis er det stor variasjon på testar. Generelt lagar B testar som skal forsøka å finna bestemte typar feil. Testar blir satt saman av mange forskjellege testteknikkar. Fokuset er å finna feil ut ifrå forventningen om kva som eignar seg best til å finna feilen. Ein test som skal verifisera [47] og validera [48] [49] eit krav kan bestå av mange testteknikkar og kan utførast i fleire testnivå (vedlegg 1). Erfaringar gjeld gjerne testar i tydinga av ”ein miks av testteknikkar”, og stor variasjon på testar gjer det vanskeleg å få skråsikre oppfatningar om ein test i ein situasjon kan samanliknast med ein test i ein annan situasjon.

Utviklingsprosessen for produktet

B har erfart at utviklingsprosessen for programvara er ein vesentleg faktor som påverkar kor mange feil ein finn i dei ulike testane. Dei seinare år har det vore ein trend mot meir agile utviklingsmetodar [2]. Agile metodar blir også kalla lettvektsmetodar eller smidige metodar. Testteknikkar utført i slike utviklingsprosesar blir gjerne kalla agil testing. Desse metodane for systemutvikling legg mellom anna vekt på god kommunikasjon, fleksibilitet og svært hyppige del-leveransar [50]. Denne trenden har ført til sterkare integrasjon av utvikling og testing. Både test- og utviklingsprosessen går parallellt i iterasjonar. Agile metodar blir vurdert å ha relativt stor vekt for å finna ulike feil, men B synest det er vanskeleg å sei om den er relativt av større vekt enn kva for testteknikkar som blir brukt. Å finna feil ved prosessar blir framhalde som ein understøttande faktor til det å finna feil i produktet. Ved agile metodar lagar utviklaren funksjonelle testar på sin kode. B erfarar at utviklarane legg like mykje arbeid i testarbeidet sitt som i kodearbeidet. B er overbevist om at dette har ført til færre feil totalt sett enn ein hadde tidlegare. Tidlegare var testarbeidet mindre prioritert av utviklarane. B har erfart at feil i koden bør helst oppdagast innan ein time etter at koden blir utvikla. Fordi det krev mykje arbeid å setja seg inn i koden ved debugging på eit seinare tidspunkt. Uavhengig av testteknikk, meinar B at det utan tvil er lettare å oppdaga feil når tankegangen for skaping av koden er fersk. I denne kritiske timen vil testverktøy som utviklarane brukar vera svært viktige.

Testverktøy

A poengterer at det har vorte meir og meir slik at ein snakkar om testverktøy i same åndedrag som ein snakkar om testteknikkar/testmetodar. Eksempler på testverktøy er JUnit [51], Selenium [52], Fit [53], FindBugs [54] og PMD [55]. Ulike testteknikkar kan vera implementert i ulike verktøy. Det blir ofte ufullstendig å snakka om testteknikkar utan å snakka om testverktøy. Når problemstillingen brukar uttrykket ”testmetoden i seg sjølv”, så er ofte ein testmetode ikkje noko som er ”i seg sjølv” i praktisk testarbeid i industrien, men integrert i eit testverktøy. Dette gjer variasjonsbreidden berre endå større enn den er for termen ”testmetode”, og følgjeleg endå vanskelegare å samanlikna eksperiment med kvarandre. A synest det er ein trend at prioritering av testaktivitetar går i retning av å leggja inn reglar for testing i eit verktøy heller enn å prioritera manuell testing.

B viser til at dei fleste IDE [56] i dag har ein kompilator innebygd som kan finna syntaktiske feil . I tillegg kan det vera snakk om funksjonalitet for å måla/testa linjedekning, forgreiningsdekning, måling av kompleksitet i koden og indikasjonar på kvar i koden potensielle feil kan liggja. Å ha ein god IDE eller eit godt testverktøy, er altså viktig, uavhengig av testteknikken. Der B arbeidar, har dei utvikla eit eige testverktøy (i Java) som kan simulera naboklassar og nabosystem, og som er til stor hjelp ved integrasjonstesting (vedlegg 1, sjå også [41]).

C har erfart at verktøy som FitNesse [57] og RSpec [58] let prosjektet uttrykka krava i form av funksjonelle testar. Dette er ein god måte å forsikra seg om at utviklarane har forstått krava, og han ser på funksjonelle testar som ein eksekverbar kravspesifikasjon.

Personleg kompetanse til å finna feil

A har erfart at uansett testteknikk, vil personen som utfører testen ha stor innverknad. Det går på både erfaring og "nase" for å finna feil.

B trekker fram forståinga av kravspesifikasjonen som ein viktig faktor. Denne forståinga er uavhengig av testteknikken.

Kompetansebakgrunn

Tilbakemelding frå testar B:

"Min erfaring er at en tester med utviklerkompetanse finner feil som testere uten denne bakgrunnen ikke finner. mao de er flinkere til å analysere seg frem til hvordan strukturen i koden er bygd opp ved hjelp av de feil som tidligere er funnet, og finner da feil på bakgrunn av dette."

Dedikert nøkkelperson til kodelesing

A har erfart at ein dedikert person til kodelesing har teke 50-60 % av alle feil. Føresetnaden er at denne personen kjenner systemet godt, har stor erfaring, er ein strukturert seniorutviklar og ikkje minst har mandat til å returnera kjeldekoden produsert av utviklaren. Dette er feil som har passert uoppdaga gjennom programmeraren sin test. Det er store kostnader å spara ved å oppdaga feil såpass tidleg (på lågnivå heller enn i systemtesting). Teknikken må, under desse føresetnadane, så absolutt kallast komplementær til einheitstesting (vedlegg 1, sjå også [40]). A har erfart at mange utviklarar har for låg terskel for å levera ifrå seg kode. Men også at når ein blir utsatt for eit effektivt kodelesingsregime, med korrektur av kodestandarder og val av felles funksjonar, går det kort tid før ein endrar adferd/holdning. Kodelesing har tradisjonelt vore sett på som kjedeleg, men A si erfaring er at med dei kostnadsinnsparingar det har ført til, har det fått høgare status og betre motivasjon for slike oppgåver. Behovet for dedikerte kodelesarar avheng av storleiken på applikasjonen. Truleg er grunnen til dette at i små utviklingsgrupper blir det lettare til at ein spør andre viss det er noko ein lurer på ved kodeskrivinga enn i store utviklingsgrupper på for eksempel 40 personar.

Innstilling til å finna feil

C meinar innstillinga testaren har til å finna feil er den viktigaste faktoren for å finna ulike feil:

”Men viktigst av alt er mindsettet. Hvis du bruker alle metoder og ønsker å finne feil, så finner du feil. Hvis en tester ikke ønsker innerst inne å finne feil, så virker produktet tilsynelatende helt perfekt! ”

A har erfart at det er også viktig kor kritisk ein er innstilt til den programkoden ein skal testa. Viss ein er ukritisk, er det hans erfaring at ein ikkje finn så mange feil som om ein er kritisk.

Rolle i testprosessen

A har erfart i konkrete prosjekt at den rolle ein person har i ein aktuell situasjon i eit testteam, har innverknad på kva for feil som blir oppdaga. For eksempel er det forskjell på om ein er elev eller profesjonell testar. Sjølv om personen er bevisst på dette, er det hans inntrykk at bevisstheit om det ikkje nøytraliserer at det blir ein forskjell.

B har erfart at fokus vil også vera ulikt frå ein utviklar til ein testar. Utviklaren har fokus på arkitektur, testaren på korleis ein skal testa krav og arkitektur.

Testnivå / testfokus

Det er B si erfaring at det fokus ein har når ein testar, spelar inn på kva for feil ein finn. Fokus vil variera etter på kva for testnivå i V-modellen [59] ein befinn seg (vedlegg 1). V-modellen er ein modell over testaktivitetar i forhold til ulike nivå i utviklingsprosessen. Same testteknikk kan brukast i ulike testnivå. B har erfart at dei same testteknikkane finn ulike feil når dei blir brukt på ulike testnivå. For eksempel kan grenseverdianalyser (vedlegg 1) brukast både i ein einheitstest og i ein akseptansetest. For einheitstesting vil fokuset kunna vera å finna feil i ein bestemt metode i ein komponent, mens fokuset for testmetoden viss testnivået er akseptansetesting (vedlegg 1, eller sjå [60]) vil kunna vera å finna feil i brukargrensesnittet.

Testbudsjett / tid til disposisjon

Testbudsjett er ifølgje B si erfaring eit vesentleg forhold som legg premissar for kor mange feil ein finn totalt. Likeeins kor mykje tid ein har til disposisjon for å finna feil ved hjelp av ulike testmetodar. Det kan vera stor variasjon på kostnadane ved bruk av ulike testteknikkar, slik at ei samanlikning av om testteknikkar er komplementære kan slå ”urettferdig” ut for dyre testteknikkar. Å testa ein nettbank-applikasjon er dyrare enn ein webapplikasjon. Ulike teknikkar kan bli prioritert ulikt avhengig av kostnadane med å debugga feila og konsekvensane av å la feila bli verande i programmet.

4 Erfaringar om testing formidla på Software 2010

Eg deltok på Software 2010 [61] i Oslo, på begge arrangementa som Faggruppa for Software testing arrangerte.

Hensikten var å få ei breidare referanseramme for dei generelle problemstillingane. I det følgjande har eg formidla erfaringar eg ser som relevante, gruppert etter kven desse erfaringane vart formidla av.

Som kapittel 2 og 3 er også dette kapittelet tenkt å gje svar på dei generelle problemstillingane:

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*
- *I kva for grad finn ulike testteknikkar ulike feil?*
- *Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?*

4.1 Erfaringar frå Lloyd Roden

Roden [62] har arbeidd i programvareindustrien sidan 1980. Han har utvikla program i Cobol, PL1, Pascal og Fortran i 5 år. I 14 år arbeidde han som testar og testleiar, og dei siste 11 åra har han arbeidd med rådgjeving innan testing.

Testleiarar og leiarar burde testa program

Synspunktet/erfaringa som kjem fram i denne overskrifta dreiar seg om organisasjonspsykologi, og er eit av andre forhold enn testmetoden i seg sjølv som har innverknad på om feil blir oppdaga.

Roden sitt synspunkt om at testleiarar og leiarar burde testa program, er ein av dei 6 største utfordringane han meinar ein står overfor i dagens testkvardag [63]. Roden hevdar dette vil gje ein testleiar større tillit i organisasjonen. Det vil vera bra for lagånda og bli oppfatta som å stå fram med eit godt eksempel. Det vil auka forståinga for god testestimering (vedlegg 1). Med testestimering forstår vi estimering av behov for tid og personar til å utføra testing. Roden har møtt på ulike unnskyldningar gjennom mange år for ikkje å kunna testa:

"I can't perform testing as I don't know the application".

"I don't have time to perform testing, I have meetings to attend, reports to write, schedules to monitor…".

"I have 32 testers reporting to me, you are not seriously suggesting that I test as well?".

”I have done testing in the past, I have now moved on. I have my team who do it much better”.

Problem med estimering av tid og personell til testing

Problem med testestimering (vedlegg 1) er ein av andre faktorar enn testmetoden i seg sjølv som har innverknad på om feil blir oppdaga.

Roden meinar at eit av problema vi ofte har med testestimering, er at den blir feil gjennomført [63]. Enkelte aktivitetar kan ikkje aksellererast ved å leggja til fleire ressursar. Han refererer til Brooks lov, og minner ironisk om at sjølv om det tek ei kvinne 9 månader å bera fram eit barn, vil det ikkje ta 9 kvinner 1 månad. Han poengterer også sterkt av vi aldri bør estimera berre for ein syklus i eit iterativt utviklingsprosjekt.

Han synest også at kvalitetsfokuset må vera sterkare i testestimering. Det er vanleg å laga estimat på grunnlag av mengde arbeid og personell. Hans erfaring er vektlegginga av av kvaliteten av programvara er sterkt undervurdert. For kvar iterasjon ein avsluttar, bør ein estimera kor mange feil som blir med over til neste iterasjon.

Roden trekte fram folks oppfatning av ordet ”estimering” i testkontekst som eit problem. Oppfatninga av ordet i sin alminnelegheit hevdar han er ”ei omtrentleg utrekning eller vurdering, basert på profesjonelt skjønn av ein ekspert”. Men i testkontekst (og utvikling av programvare) hevdar han det blir brukt synonymt med anbod (i kontrast til eit fastpris-tilbod).

Roden såg fullt av paradokser i den praktiske testkvardagen. For eksempel, dess meir tid ein brukar på å rapportera feil, dess mindre tid vil det bli att for å finna feil. På den andre sida er optimal testing avhengig av gode estimat, og dess betre ein dokumenterer feil dess betre grunnlag har ein for å bruka historiske data for å laga framtidige estimat.

Det er også eit paradoks at dess meir feil ein finn, dess større er behovet for å testa programmet, men dess mindre blir tida for å testa det.

Eit tredje paradoks er at estimat sannsynlegvis blir betre viss dei kan utførast av dei som skal gjera testarbeidet, men i praksis er det ofte slik at estimat blir laga av testleiaren.

Eit fjerde paradoks er at tidlege estimat er unøyakige men har stor innverknad på testarbeidet, mens seine estimat er nøyaktige, men er mindre viktige for testarbeidet.

Andre forhold som er viktige for testresultatet

Sjølv forutsatt rett bruk av termen estimering, hevda Roden [64] at estimering er vanskeleg, og dei grunnane han trekte fram for det er samtidig ei liste over ”andre forhold som er viktige for testresultatet”:

- Oppgåver kan vera vanskelege å identifisera i ein risikoanalyse.
- Det kan vera stor uvisse knytt til kva for personar ein kan rekna med er disponert til testaktivitetar til eikvar tid i eit prosjekt.
- Det kan variera kva for kompetanse dei som skal testa har.
- Krava til programmet som skal testast kan vera vage (eller delvis manglande).

- Krava kan bli gjeve lov til å auka undervegs. Fenomenet er kjent som ”scope creep” [65].

- Frist for ein leveranse kan i realiteten ofte vera bestemt på førehand, slik at estimering reelt sett ikkje er så viktig.

- Estimat kan bli satt overoptimistisk låge av vikarierande motiver, for eksempel at ein ikkje vil ta sjansen på at sjefen skal bli skuffa.

- Kvaliteten på programmet som skal testast kan vera varierande.

- Det vil på førehand vera ukjent kor mange iterasjonar testinga vil måtta trenga. På Software 2010 kom det fram synspunkt frå ein testar/testleiar som sa at ho aldri hadde vore med på eit testoppdrag der det ikkje hadde vore bruk for meir enn 1 iterasjon. Spørsmålet er i praksis ikkje om det er 1 eller fleire iterasjonar, men kor mange det vil bli. Behov for regresjonstesting [66] (sjå også vedlegg 1) etter feilsøking og feilretting i implementeringsfasen fører fort til mange iterasjonar. Dei feila som ikkje blir oppdaga i ein iterasjon, vil ein dra med seg til neste iterasjon. I tillegg kjem dei feila som blir introdusert i programmet ved retting av dei feila ein oppdaga.

- Testmiljøet er for dårleg (for dårlege testverktøy eller at konfigureringa ikkje er god nok).

- Teknologi som blir brukt kan vera ny eller ukjent for dei som skal bruka den.

- Viss tilliten til dei testane og testscripta ein har er stor, kan det vera ei ulempe (innstilling til å finna feil).

Altfor komplekse program

Roden [64] tilrår å utfordra komplekse kravspesifikasjonar og design dokument ved eitkvart høve, og vurdera kritisk kva som verkeleg er nødvendig å ha. Han er skeptisk til den trend han ser i tida med at det som er enkelt blir sett på som uinteressant, mens det som er komplisert blir sett på som ”godt” og ”interessant”. Roden refererte til Jim Johnsen [67] om at kun 25% av all funksjonalitet i programvare eigentleg blir brukt, at 45% aldri blir brukt og at kun 20% blir brukt ofte.

Vurder på bakgrunn av kontekst

Essensen i dei råd Roden gav [64] [63] er at ved testestimering (vedlegg 1), akkurat som ved testing, handlar det om å auka sin bevissheit om at det er kvalitet det handlar om alt saman.

Ifølgje Roden treng vi å forstå dei faktorar som gjer testestimering så vanskeleg, for å gjera så gode val som råd. Ein bør skifta estimeringsteknikk alt etter kva for situasjon ein er i. Overført til dei skulane som er for testing (sjå kapittel 6) oppfatta eg Roden som talsperson for den kontekst-drivne skulen.

4.2 Erfaringar frå Julie Gardiner

Julie Gardiner [68] blir presentert mellom anna slik på nettsida til firmaet Grove Consultants der ho arbeidar:

"Julie started work in the IT industry in 1991. She worked as an analyst programmer, an Oracle data base administrator and as a Project Manager. This work gave Julie first hand experience of the roles of test analyst, test team leader, test consultant and test manager. Her experience has been gained across a broad range of industries including financial, utilities, retail, insurance, construction and the public sector. She has used various software development approaches from the traditional to agile methodologies."

Testmiljø

Gardiner er tydeleg på at utan eit godt testmiljø, har vi ingen sjanse til å testa. Viss testmiljøet er dårleg, vil det også gå ut over estimeringa.

Prefererte testteknikkar ved systemtesting og akseptansetesting

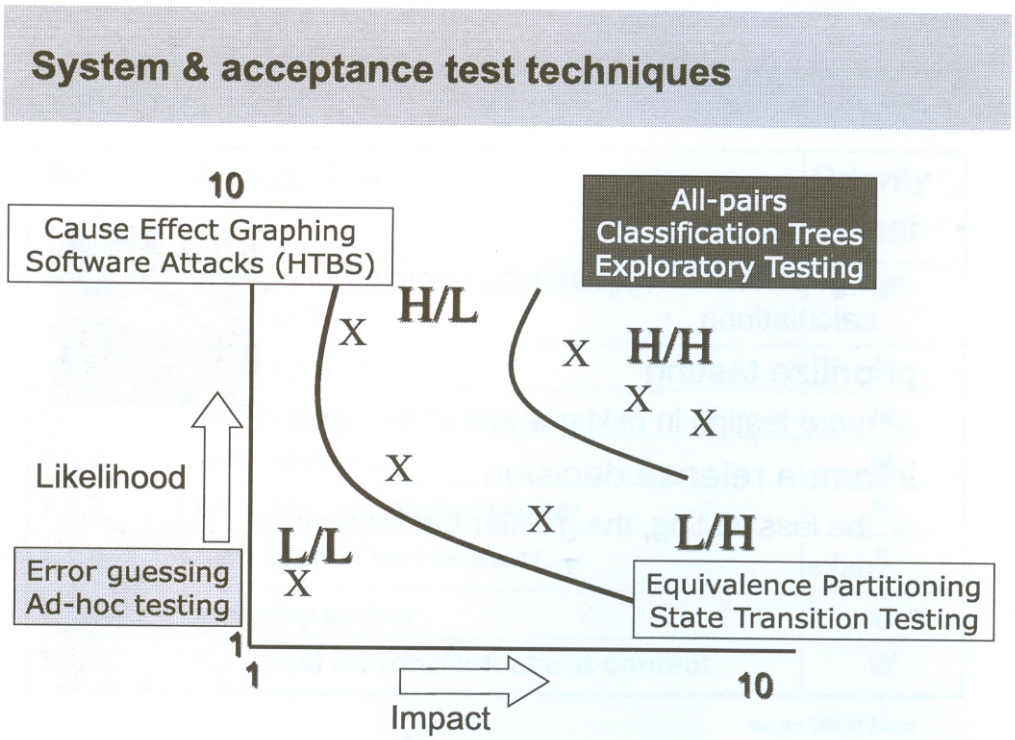
Julie Gardiner [69] poengterte at testing og risikovurdering er to sider av same sak.

Den risiko programfeil inneber er produktet av sannsynlegheiten for at feilen skal inntreffa og konsekvensane av feilen.

I eit XY-diagram (sjå figur på neste side) plasserer ho øverst i høgre kvadrant testteknikkane ekvivalenspartisjonering (vedlegg 1, sjå også [70]), grenseverdianalyse og beslutningstre [71] som hennar favoritt-teknikkar når det gjeld testnivåa einheitstesting og integrasjonstesting. Øvre høgre kvadrant er testteknikk med størst sannsynlegheit for å oppdaga feil som har stor sannsynlegheit for å forekomma og som får store konsekvensar. Altså høgrisiko-feil.

For systemtesting og akseptansetesting er testteknikkane parvis testing [72], klassifikasjonstre (vedlegg 1) og utforskande testing hennar favorittar, som også følgjeleg er plassert i øvre høgre kvadrant.

Det var interessant å observera at ho plasserte testteknikken ad hoc testing (vedlegg 1, eller sjå [73]) i nederste venstre kvadrant (minst risiko for at feilen vil skje og minst konsekvensar). Den indirekte informasjon som ligg i dette er at ho ser på utforskande testing og ad hoc testing som to ulike teknikkar.



Figur 1: Testteknikkar for systemtesting og akseptansetesting, preferansar frå Julie Gardiner

Ho har observert gjennom eigen erfaring at risikovurderingar i testarbeidet har auka vektlegginga av testing og har ført til sterkare samarbeid mellom dei som utviklar koden og dei som testar den.

Risikobasert tilnærming

Dess mindre testing, dess større risiko gjenstår. Til sjuande og sist er det forretningssida som må ta risikoen. Risiko endrar seg over tid i eit utviklingsprosjekt, difor er det nødvendig å ha ei fleksibel tilnærming. Risikofokusert argumentasjon bør brukast overfor leiinga i bedrifta for å få aksept for tilstrekkeleg testing. "Forget the deadline", seier Julie Gardiner. Ho vektlegg også "The language of risk", og ho har erfart at i 80% av tilfella har ho oppnådd å få forlenga sluttdatoen for testing ved å få leiinga til å høyra på argumentasjonen og ta den på alvor. Måten ein testar stiller spørsmål på, kan vera viktig. Det kan vera eit sterkare "argument" å spørja: "Kva er det du *ikkje* vil vi skal testa?" enn å spørja: "Kva er det du vil vi skal testa?".

4.3 Erfaringar frå Telenor

Etter å ha presentert erfaringar frå to privatpersonar på Software 2010, følgjer no nokre norske firma sine erfaringar. Det blir ikkje differensiert på om erfaringane gjeld den/dei personar som formidla erfaringane eller om dei gjeld for firmaet som sådan.

Telenor har frå 2005 hatt ei felles testavdeling, med felles testmetodikk, testprosessar og malverk [74]. I 2008 vart avdelinga delt mellom mobiltelefoni og fasttelefoni. Tilsaman arbeidar det ca. 40 personar i testavdelinga.

Det er ofte mange system som blir utvikla parallellt, dette gjer det ofte vanskeleg å bruka lettvektsmetodar i tilnærmingen til utvikling og testing. Ein har difor valt å arbeida etter fossefallsmetoden i dei fleste samanhengar.

Avvik frå tidsplanen i tide

Ei generell erfaring er at når ein ser at tidsplanen vil sprekka, unnlet ein å ta konsekvensen av det i tide, men håpar heller på eit "mirakel". Erfaring tilseier at det skjer ikkje alltid at ein seier "stopp" når ein burde, for å forbetra kvaliteten før ein går vidare.

Tidleg brukarinvolvering

Ofte kjem ikkje brukarane av systema med før i slutten av prosjektet, i testfasen. Dei bør komma med tidleg.

Testmiljø

Det er viktig for kvaliteten av testarbeidet at det tekniske testmiljøet er godt.

Graden av kvalitetskontroll undervegs

Ein har ofte erfart at det har vore for lite kontroll på kvaliteten i software-prosjekt undervegs. Kvalitetskontrollen bør gå både på produktet og prosjektaktivitetar, i form av planlagte/systematiske kvalitetskontrollerande aktivitetar. Dette vil vera risiko-reduserande. I Telenor ser ein på test som ein del av kvalitetskontrollen. Ein brukar gjerne definera test som "måling av kvalitet". Ein erkjenner at ein kan aldri finna alle feil. Testing blir sett på som ein metode for risikoreduksjon, ikkje risikofjerning. Ved å ha systematiske kvalitetskontrollar undervegs har ein erfart å oppdaga ein god del feil tidleg. Dess tidlegare feil blir oppdaga, dess billegare er det å retta dei. Ved å ha fokus på kvalitet gjennom heile prosjektforløpet, oppnår ein vesentlege gevinstar på test-effektivisering. For eksempel å oppdaga at feil kan skjula andre feil, eller at feilretting kan introdusera nye feil.

Vel form for statisk testing basert på kontekst

Med statisk testing [75] forstår ein gjennomgang av dokumentasjon. Ulike former for gjennomganger er distribusjon, til uttale (du sender noko ifrå deg og ber om tilbakemelding på det), uformelle gjennomgangar og teknisk formelle gjennomgangar. Val av gjennomganger avheng av bedriftskultur, risiko i prosjektet og tilgjengeleg tid.

Rollar i testprosessen

I Telenor si testavdeling er det 5 rollar for gjennomføring av statisk testing: Testleiar, fasilitator, forfattar, ekspert og sekretær.

4.4 Erfaringar frå Sogeti

Sogeti er eit søsterselskap til Capgemini, med 50 tilsette i Norge. Selskapet er globalt, og har ca. 20.000 tilsette i 15 land. Selskapet arbeidar dedikert med testing og testleiing. Ein brukar dei standardiserte metodane TMap og TPI til testaktivitetar [76].

Bruk risikobasert tilnærming

Det er viktig med kjennskap til risiko i eit prosjekt for utvikling av programvare. Risiko knytt både til produktet og prosessen. Produktrisiko er risikoen for kunden dersom systemet ikkje tilfredsstiller forventningane. Det kan vera teknisk risiko ved at ein komponent ikkje er robust nok, eller det kan vera forretningsrisiko ved at bedrifta sitt rykte i marknaden blir skada.

Risikovurderingar ligg til grunn for kva ein skal fokusera testinga på, kva for testteknikkar som skal brukast. Risikoområder i programmet bør testast først. For gradering av risiko brukar ein uttrykket risikoklasse. Risikoklasse er produktet av sannsynlegheiten for at feil vil inntreffa og konsekvensen av feilen.

Risikoklasse = sannsynlegheit for å finna feil * konsekvens
= (brukshyppigheit * sannsynlegheit for feil) * konsekvens.

Rådet er å testa dei største risikoområdene først. Dette meinar ein gjev betre tid til å retta feil, og skaper større ro og tryggleik i eit prosjektet.

La kunden styra testprosessen

Testaren si rolle er å framskaffa grunnlag for at kunden kan ta avgjerder. Parametrane er resultat, risiko, tid og kostnad. Det er alltid ein eller fleire av desse faktorane det er fokus på. Ein ser på informasjon som det viktigaste resultat frå testaktivitetar. At kunden avgjer vektlegginga av desse fire parametrane kallast forretningsdriven testleiing. Det blir også kalla retningsbasert testleiing.

Testplanlegging

Sogeti brukar ein standardisert og strukturert metodikk (TMap) som hjelp til å planleggja testinga. Det kan vera vanskeleg å vita korleis ein skal starta eit testarbeid. Metoden er meint å gjera starten lettare. Sjølv om metoden er standardisert, kan den tilpassast etter behov (kontekst) basert på vurdering av systemet sin kompleksitet, produktrisiko, avhengigheit til andre system / prosjekt. Val av kombinasjon av testteknikkar må alltid bli ei vurdering i stor grad basert på skjønn og erfaring.

Ei erfaring som vart formidla var at det går (nesten) aldri som planlagt i ein testplan. Dei vanlegaste feila er at kodinga dreg ut i tid, at kvaliteten er for dårleg på koden eller at testmiljøet ikkje er godt nok. Difor må det planleggjast på nytt, og ein må fokusera på det som er viktigast. At utviklingsarbeid overskrider tida og at kunden står på forventningskrava sine, gjer at den som testar ofte kjem i ei klemme. At det ofte er sluttbrukar som finn feila, medverkar også negativt.

4.5 Erfaringar frå Statens Pensjonskasse

Erfaringane som vart formidla på Software 2010 gjaldt prosjektet PERFORM, der dei utviklar og vidareutviklar programvare for å imøtekomma dei nye krava i pensjonsreforma [77]. Endringar i dette regelverket, som trer i kraft frå 1. januar 2011, vil medføra behov for endringar og tilpasningar av eigne system og arbeidsprosessar. Det er 350 faste medarbeidarar i Statens Pensjonskasse. Ein satsar på stor grad av intern utvikling og vedlikehold av programvare. Det er ca. 180 personar på fulltid i PERFORM-prosjektet, og ca. 70 av dei tilsette arbeidar på fulltid med dette. I tillegg har ein 130 innleigde konsulentar. Alle sit på ei golvflate. Som utviklingsmetodikk brukar ein SCRUM [78], som er ein iterativ lettvektsmetodikk. Til saman er det 11 scrumteam.

Kontinuerleg systemintegrasjonstesting

PERFORM-prosjektet starta i 2008 og er estimert å vara til 2011. Det er 3 hovudleveransar i året, og målet er at PERFORM skal levera til produksjon i hovudleveransane. Denne gradvise innfasinga er risikoreduserande. Kritisk funksjonalitet blir delt opp mellom ulike hovudleveranser. Ein får tilbakemeldingar frå verkeleg bruk av løysinga undervegs. Ein oppnår tidleg gevinst av ny funksjonalitet. Dette fører til at konstruksjon (design og programmering) pågår kontinuerleg, fordi prosjektet er i tre leveransar samtidig. Ein har erfart at levert funksjonalitet blir endra på nytt, og det er viktig å ta høgde for dette i estimeringa. Ein ser på funksjonell testing og integrasjonstesting som ein del av konstruksjonen. Testing og utvikling foregår parallellt. Kvart scrumteam testar sin eigen kode.

Rollar i testprosessen

Kvart scrumteam har 7-9 personar, og rollane er funksjonelt ansvarleg, juniorutviklar, seniorutviklar, teamarkitekt, testar og scrum-master.

Godt testmiljø er viktig

Når kvart testteam skal testa sin eigen kode, krever det eit godt testmiljø innan kvart team. Det krev automatiserte testar, og PERFORM har brukt FitNesse [57] som testverktøy. Det har vore delte erfaringar med FitNesse. FitNesse overtok for JUnit [51] for einheitstesting i 2008, og det har ført til mindre bruk av JUnit. JUnit er eit verktøy for einheitstesting i Java. Med FitNesse har ein erfart at enkle endringar i løysningen har ført til vedlikehold av over 300 testar. Dette har redusert farten i utviklingsarbeidet. I 2009 laga ein eit rammeverk med malar for FitNesse som gjorde at ein automatisk fekk fram alle inndata og alle utdata utan at ein testa på dei. Dette medførte også at testarane igjen fekk kjensle av eigarskap til testane. Det førte også til at refaktorering [79] vart forenkla og at ein ikkje vart så redd for å røra kompleks kode.

Det krevest eit systemintegrasjonsmiljø når alle 11 scrumteam samlar alt ein har gjort i dei siste iterasjonane og gjennomfører testar på tvers av systema og på tvers av dei ulike scrumteama. Eit kontrollpunkt før ein hovudleveranse er at all utvikla funksjonalitet må vera køyrt i integrasjonstestmiljøet. Kvaliteten på produktet i desse testane må verifiserast og godkjennast av produkteigaren.

4.6 Erfaringar frå Storebrand

Testeffektivisering kan vera motivert av eit ønskje om å gjera ting raskare, betre og billegare [80]. Ønskjet kan komma både frå leiinga og frå fagpersonell. Frå leiinga kan stikkord for motivasjonen vera finanskrise, kostnadskutt, konkurransesituasjonen eller nye offentlege pålegg. Frå fagpersonell kan motivsjonen vera for dårleg kjennskap til produktkvaliteten, for dårleg teknisk testmiljø og/eller testdata, at ein ikkje finn alvorlege feil tidleg eller at feil ikkje rettast fort nok slik at ein får problem med å rekka tidsfristar.

Storebrand formidla erfaringar med bruk av rammeverket TPI (Test Process Improvement). Metoden TPI Next [81] er ein metodikk varemerkt av Sogeti. Metodikken er erfaringsbasert, og utvikla og forbetra gjennom 15 år.

TPI Next som rammeverk for testeffektivisering

TPI Next identifiserer 16 nøkkelområder for effektivisering, og grupperer desse i tre grupper. Kulepunkta under viser desse gruppene og underkulepunkta viser dei 16 nøkkelområda:

- Involvering frå interessentar
    - Graden av forplikting frå interessentar
    - Graden av involvering/engasjement
    - Teststrategi
    - Testorganisasjon
    - Kommuniksjon
    - Rapportering
- Testleiing
    - Test prosess
    - Estimering og planlegging
    - Målemetodar
    - Feilrettingstid
    - Dokumentasjon av testinga
- Testfagleg
    - Metodologi, praksis, gjenbruk
    - Testkompetanse
    - Design av testtilfelle

- Testverktøy
- Testmiljø

Storebrand sine erfaringar er at dette rammeverktøyet har hjelp dei med å finna fram til kva dei bør og kan effektivisera. Metodikken TPI Next blir brukt til å måla modenheitsnivået organisasjonen har innanfor nøkkelområda.

5 Diskusjon av resultat så langt

I dette kapittelet konkluderer eg kva masteroppgåva så langt har kunna frambringa av funn, og evaluerer korleis funn samsvarar – eller ikkje samsvarar – med kvarandre. Eg søker etter eventuelle mønster som avteiknar seg som kan gje svar på dei generelle problemstillingane:

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*
- *I kva grad er ulike testteknikkar komplementære når det gjeld å finna feil?*
- *Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?*

5.1 Diskusjon av generell problemstilling nr. 1

Problemstilling 1 lyder: Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?

Artikkel 1 kårar ingen klar "vinnar" av testteknikkane funksjonell testing, strukturell testing og kodegjennomgang. Artikkel 2 finn at funksjonell testing, strukturell testing og inspeksjon har ulike styrke og svakheit, utan at ein tolkar det som om den eine er "betre" enn den andre. Artikkel 3 samanliknar inspeksjon og testing og finn at inspeksjon er meir effektiv testteknikk i designfasen enn testing. Dette funnet blir støtta av artikkel 5, som hevdar inspeksjon er meir effektiv enn testing både for å finna feil i krava og i designet.

Artikkel 5 konkluderer med at for kodefasen er det mange studiar som meinar at funksjonell og strukturell testing er meir effektive enn inspeksjon.

Julie Gardiner har erfart at for systemtesting og akseptansetesting er testteknikkane parvis testing, klassifikasjonstre og utforskande testing dei mest veleigna, mens ad hoc testing er den minst effektive testteknikken. Dette blir delvis støtta av testar B som har erfart at testteknikkane utforskande testing og destruktiv testing utmerkar seg ved å finna andre feil enn dei spesifiserte testtilfella påviser.

Testar B har erfart at det ofte hender at testteknikkar finn andre typar feil enn dei ein forventar. Uventa feil blir ofte funne av funksjonelle testar. Testar B har erfart at dette ofte er det tidlegaste i utviklingsforløpet at åpenbare feil kan bli oppdaga. Generelt talar ei slik erfaring for at ein bør ha ei vid ramme for kva testteknikkar ein brukar, og ikkje vera altfor forutinntatt til kva feil dei vil avdekka.

Testar B har erfart at inspeksjonar eignar seg godt til å oppdaga logiske feil. I tillegg medverkar dei indirekte til betre vedlikehold ved at dei skaper større forståing for kor viktig det er med felles standarder for programmeringsstil.

5.2 Diskusjon av generell problemstilling nr. 2

Problemstilling 2 lyder: I kva grad er ulike testteknikkar komplementære når det gjeld å finna feil?

Det synest vera større tru på at ulike testmetodar er komplementære hos dei testarane eg har snakka med frå industrien enn det som kan utleiast frå dei 5 artiklane.

Men bildet er ikkje eintydig. Testar B synest det er vanskeleg å ha sikker meining om inspeksjon og dynamisk testing er komplementære. Hypotesen er at ulike metodar finn ulike feil. Men i ein travel kvardag blir det ikkje tid til å få verifisert det inntrykket han har av at ulike metodar finn ulike feil.

Testar C meinar at lågnivå integrasjonstesting finn mange feil som einheitstesting ikkje finn. Han meinar også at feil i krava vil oppdagast kun i brukartestar. Og at feil i logikken i skjermbilder vil oppdagast i lågnivå integrasjonstestar, funksjonelle testar eller brukartestar. Feil i bruk av infrastruktur vil oppdagast i integrasjonstestar, enten på lågnivå eller systemnivå. Det er mykje billegare å finna desse feila på lågnivå. Systemnivå intergrasjonstestar er einaste måten å finna feil i produksjonsaktig konfigurasjon, nettverksproblemer etc. Feil i utviklarens forståing av krava kan oppdagast av funksjonelle testar eller brukartestar. Dei blir mest effektivt oppdaga i funksjonelle testar.

Artikkel 4 konkluderer med at strukturell testing og kodeinspeksjon i liten grad er komplementære. Dei finn med andre ord dei same feila. Indirekte er dette motstridande til konklusjonen i artikkel 2 om at kombinasjon av fleire testteknikkar aukar talet på feil ein finn.

Testar C har erfart at einheitstestar ofte finn feil som utviklaren sjølv finn på å testa for. Sidan idear til slike testar er individuelle, vil dei sannsynlegvis variera frå utviklar til utviklar, og difor vil testteknikken i stor grad bli komplementær til andre testar. Erfaringa om at dette er ein kostnadseffektiv måte å testa på, styrker denne trua.

5.3 Diskusjon av generell problemstilling nr. 3

Problemstilling 3 lyder: Er oppdaging av feil avhengig av testmetoden i seg sjølv, eller er det andre faktorar som spelar inn?

Testmetoden i seg sjølv

Det framgår tydelegare frå samtalane med testarane frå industrien enn frå dei 5 artiklane at ein er tilbøyeleg til å meina at feila som identifiserast er avhengige av testmetoden i seg sjølv. Testar A påpeikar at det ligg i testteknikkane sin natur å oppdaga ulike feil, og han refererer til 5 konkrete eksempler på det. Men både testar A og testar B har reservasjonar. Testar A sin reservasjon er at han kan ikkje sjå for seg at det er praktisk mogeleg for industrien å frambringa statistisk materiale for erfaringane deira. Testar B sin reservasjon går på at det er testtilfella i minst like stor grad som testmetodane som er viktige for kva for feil som blir oppdaga. Testar B trekker som reservasjon også fram at einheitstestar finn feil som utviklaren kjem på.

Det er ingen av dei refererte artiklane som påstår at feila som blir identifisert kun er avhengige av testmetoden i seg sjølv.

Utføringen av ein testteknikk

Artikkel 5 trekker fram at det like gjerne kan vera utføringa av ein testteknikk som testteknikken i seg sjølv som medverkar til om feil blir funne. Testteknikkar er ikkje så stringent definert at det ikkje er rom for fleire måtar å testa på sjølv om ein brukar same term-etikett på det ein held på med.

Utviklingsmetodikk

Testar B har erfart at lettvektsmetodikkar finn feil meir effektivt enn meir tradisjonelle fossefallsmetodikkar. Testar B har erfart at med lettvektsmetodikkar legg utviklarane like mykje arbeid i å testa koden som å utvikla den. Dette har ført til færre feil enn før.

Lloyd Roden påpeikar at det er eit ukjent behov for tal på iterasjonar i lettvektsmetodikkar, eller i utvikling generelt for den del. Talet kan auka som følgje av introduksjon av feil ved retting av feil. For Statens Pensjonskasse set det å driva kontinuerleg systemintegrasjon store krav til rett og felles bruk av utviklingsmetodikk. Storebrand har testprosess som eit av nøkkelområda i metoden TPI Next for testeffektivisering.

Testverktøyet

Testar B har erfart at feil bør helst oppdagast i løpet av ein time etter at den er introdusert i koden. Då blir testverktøyet den enkelte testar sjølv har til disposisjon, svært viktig.

Testar A har erfart at verktøy og testteknikk i aukande grad har vorte nemnd i same åndedrag i løpet av dei siste åra. Dette har vore ein trend han har merka. Ulike testteknikkar kan vera implementert i ulike verktøy. Han har observert ein trend med å leggja inn reglar for testing i eit testverktøy heller enn å prioritera manuell testing. Testar B kompletterer dette bildet ved å påpeika at dei fleste IDE [56] har innebygd kompilator som kan finna syntaktiske feil i koden. Testar C refererer til testverktøya FitNesse [57] og RSpec [58] som let prosjekta uttrykka krava i form av funksjonelle testar. Funksjonelle testar blir ein eksekverbar kravspesifikasjon.

Programmet sin natur

Artikkel 1 viser at nokre program er lettare å oppdaga feil i enn andre program. Også artikkel 2 viser at det er programmet sin natur heller enn typen feil som avgjer om feilen er lett å finna. Artikkel 5 påstår også at programmet som feilen finst i, kan vera viktig for kor lett feilen er å finna. Det er nærliggjande å tru at dette kan ha med kompleksiteten å gjera. Lloyd Roden poengterer at for mange program no for tida har altfor komplekse kravspesifikasjonar. I komplekse program vil det vera vanskelegare å oppdaga feil enn i enkle program.

Den enkelte feil sin natur

Av artikkel 1 framgår at det er den enkelte feil heller enn den type/kategori feilen måtte vera klassifisert under, som avgjer om feilen er lett å oppdaga eller ikkje. Dette funn vert støtta av artikkel 2 og artikkel 5.

Kvaliteten på programmet

Å få kontroll over kvaliteten på programmet er noko dei fleste er opptekne av. Lloyd Roden påpeikar at ukjent eller for dårleg kvalitet er ein grunn til at feil ikkje blir funne, eller at det kan bli vanskeleg å retta feil utan stor sannsynlegheit for å introdusera nye feil. Telenor

fokuserer på graden av kvalitetskontroll undervegs i utviklinga som ein av dei viktigaste suksessfaktorane.

Estimering av testaktivitetar og testplanlegging

Julie Gardiner ser estimering og testbudsjett som to sider av same sak. Testbudsjett legg viktige premisser for kor mange feil ein kan finna, som testar B har erfart. Gardiner veljer å innta ein offensiv holdning til trusselen om for små budsjett, risikobasert tilnærming som ho sjølv kallar det ("The language of risk"). Det same gjer Sogeti. Lloyd Roden påpeikar at det er viktig at det blir klart kommunisert at estimering ikkje er synonymt med anbod. Han hevdar at estimeringsteknikk bør veljast og endrast etter skiftande behov, også innanfor eit og same utviklingsprosjekt. Telenor gjev uttrykk for det same på ein annan måte, nemleg at det er viktig å avvika frå testplanen i tide når ein ser at eit estimat ikkje held. Slik Lloyd Roden har lært programvareindustrien å kjenna, kan frist for ein leveranse ofte vera bestemt på førehand, slik at estimering reelt sett ikkje er så viktig. Det kan vera vikarierande motiver for å setja estimat urealistisk optimistisk. Eit anna forhold han åtvarar mot, er krav som blir gjeve høve til å veksa urimeleg mykje undervegs. Då er gjerne problemet at på grunn av for vage eller mangelfulle krav i utgangspunktet har ein for dårleg informasjonsgrunnlag til å foreta risikovurderingar. Sogeti planlegg med TMap som metode, men har erfart at det (nesten) aldri blir som ein planlegg. Også Storebrand brukar TPI Next som har estimering og testplanlegging som eige nøkkelområde for testeffektivisering.

Testerfaring og testkompetanse

Artikkel 1 framheld testerfaring og testkompetanse som like viktig som testteknikken i seg sjølv. Dette blir støtta av testar A som meinar at testerfaring og "nase" for å finna feil er ein viktig faktor. Testar B held fram forståing av kravspesifikasjonen som ein viktig faktor. Dette kan tolkast som i kva for grad ein testar forstår forretningslogikken bak programmet som blir testa. Testar B poengterer også at det er viktig at den som testar har bakgrunn som utviklar. At Lloyd Roden erkjenner at kompetanse er viktig, kjem indirekte til uttrykk ved at han trekker fram varierande kompetanse hos testpersonell som ein variabel å ta hensyn til i risikovurderingar. Artikkel 5 er også inne på at kven som utfører testinga er viktig for kor mange feil som blir oppdaga.

Testmiljø

Både Lloyd Roden, Julie Gardiner og Telenor framhevar eit godt teknisk testmiljø (konfigurering av databasar og program til bruk for testinga) som viktig for kvaliteten av testarbeidet. Statens pensjonskasse påpeikar at eit godt teknisk testmiljø er viktig både for testing innan kvart scrumteam og for integrasjonstesting for delsystem utvikla på tvers av fleire scrumteam.

Innstilling til å finna feil

Artikkel 5 konkluderer med at formålet med testinga spelar ei rolle for om og kva for feil ein finn. Viss ein ser etter noko spesielt, er det større sjanse for at ein vil finna det enn om feilsøkinga har eit generelt preg. Testar C har erfart at viss du har ei kritisk innstilling til programkoden og vil finna feil, så finn du feil. Denne erfaringa blir stadfesta av testar A.

Innstillingen kan også påverkast av kva for testnivå testinga gjeld. Testar B har erfart at dei same testteknikkane finn ulike feil når dei blir brukt på ulike testnivå i V-modellen (vedlegg 1).

Programmeringsstil

Testar B har erfart at inspeksjonar medverkar indirekte til betre vedlikehold ved at dei skaper større forståing for at felles kodestandarder for programmeringsstil er viktig. Eg tolkar testar B slik at han her indirekte seier at programmeringsstil er viktig for kor lett det er å feilsøka og retta feil i eit program.

Brukarinvolvering

Artikkel 5 konkluderer med at det er viktig å leggja vekt på feilfinning blant sluttbrukarar, fordi erfaring viser at det er der mange feil blir funne. Det er også naturleg å sjå dette i samanheng med testar C si erfaring om at feil i utviklarens forståing av krava kan oppdagast av funksjonelle testar eller brukartestar. Etter testar C si erfaring blir desse mest effektivt oppdaga i funksjonelle testar. Men om det kan vera på grunn av dårleg brukarinvolvering eller til tross for god brukarinvolvering, er det ikkje holdepunkt for å meina noko sikkert om. Telenor legg vekt på tidleg brukarinvolvering, og Sogeti legg opp til å la kunden styra testprosessen.

Rollar i testprosessen

Den rolle ein har i ein testprosess kan påverka sannsynlegheiten for at feil blir oppdaga. Det er kanskje ikkje rollen i seg sjølv som er viktig, men den testkompetanse ein innehar. Testar A påpeikar at ein profesjonell testar vil finna fleire feil enn ein student. Rollen i seg sjølv kan vera viktig, fordi den kan reflektera ulike fokus. Testar B påpeikar for eksempel at ein utviklar og ein testar kan ha ulikt fokus. Telenor har 4 rollar for gjennomføring av statisk testing. Statens pensjonskasse har 7-9 personar med dedikerte rollar i kvart scrumteam. Rollane representerer rollespesialisering og arbeidsdeling. Ein annan effekt rollar kan ha, er å skapa motivasjon og lagånd. Lloyd Roden ser det slik at leiarar og testleiarar bør ta del i testaktivitetar ut frå slike hensyn.

Testar A har gode erfaringar med innføring av ein spesialrolle, nemleg dedikert nøkkelperson til kodelesing. Hans erfaring er at mange utviklarar har for låg terskel for å levera ifrå seg kode. Under optimale forhold (personen har testerfaring og kjenner godt systemet som blir testa og har mandat til å returnera kjeldekoden) er tiltaket komplementært til einheitstesting. Men behovet bør vurderast, blant anna vil det avhenga av storleiken på systemet som blir testa.

Kontekst

Lloyd Roden påpeiker at teknikk for estimering av testarbeid bør tilpassast den konkrete situasjonen til eikvar tid. Telenor tilrår val av form for statisk testing basert på kontekst. Sogeti poengterer at metoden TMap kan tilpassast etter behov basert på systemet sin kompleksitet, produktrisiko og avhengigheit til andre system/prosjekt.

6 Testing – eit problemområde med mange kontekstar

Dei føregåande kapittel har hatt som bakgrunn analyse av artiklar, praktiske erfaringar av erfarne testarar frå industrien og råd og erfaringar formidla på Software 2010 av personar med lang praktisk erfaring med testing.

Dette kapittelet søker å utvida referanseramma ytterlegare, ved igjen å søka teori i litteraturen. Eg brukar den same metoden som er brukt for kapittel 2. Teorigrunnlaget hellar meir over til samfunnsvitenskap enn naturvitenskap. Tolkning av verdisyn og holdningar set større krav til skjønn enn vurdering av meir objektivt målbare parametrar.

Testing av programvare er ikkje eit nytt fenomen. Det har iallfall eksistert sidan 1960-talet, eller så lenge som dataprogram har eksistert. Har det vore ulike syn på testing som er historisk betinga? Kan vi læra noko av historien?

Dette kapittelet er både generelt og konkret på same tid. Fordi uttrykket "kontekst" kan forklarast som eit generelt fenomen. Men det kan også forklara den konkrete konteksten for testing av Arkade. Som følgje av denne dualismen, vil den av dei generelle problemstillingane eg håpar å få svar på, vera:

- *Korleis bør eit dataprogram generelt testast og feilsøkast for mest effektivt å finna feil?*

Og den av dei spesielle problemstillingane eg håpar å få svar på, er:

- *Korleis bør Arkivverket sitt SAS-program Arkade testast?*

I dette kapittelet skal vi sjå nærmare på følgjande kontekstar:

- Kontekst 1: Forskningsparadigme
- Kontekst 2: Organisering og leiing av testarbeid
- Kontekst 3: Politikk / verdiar / holdningar / skular
- Kontekst 4: Type applikasjon

6.1 Kontekst 1: Forskningsparadigme

Tilnærminga i dette kapittel byggjer på ei erkjenning av at problemområdet 'testing av programvare' ikkje eine og åleine teoretisk kan funderast i eit strikt naturvitenskapeleg forskningsparadigme. Innanfor Software engineering har metodar med opphav frå samfunnsvitenskap i mange år vore viktige for å få fram kunnskap og erfaringar.

Det kan tyda på at Dilorio er av same oppfatning [82]:

"Effective problem solving begins with understanding that debugging is part instinct and technique, part coding. This section describes the some of art, instinct, and behavior that drive proper use of debugging tools."

6.2 Kontekst 2: Organisering og leiing av testarbeid

Det kan vera store skilnader på organisering av testinga mellom ulike firma. Ein vanleg variabel kan vera storleiken på firmaet, eller storleiken på programmet, eller begge deler. Hetzel [83] reflekterer over at omorganisering sjelden løyser dei verkelege, underliggjande problema det er meint å løysa. Han gjev eit inntrykk av at dette er eit allmenngyldig fenomen, med lange tidsperspektiv ved å referera til eit 2200 år gammalt sitat av Petronius Arbiter, 210 B.C. :

"We trained hard…but it seemed that every time we were beginning to form up into teams we would be reorganized…I was to learn later in life that we meet any new situation by reorganizing, and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency, and demoralization".

Ifølgje Brooks [1] bør hensikten med organisering generelt vera å avgrensa behovet for kommunikasjon. Han påpeikar det faktum at viss n personar deltek i eit prosjekt, er det $(n^2-n)/2$ grenseflater for kommunikasjon. Dette er vist i tabellen under:

Talet på personar i eit prosjekt	**Talet på grenseflater for kommunikasjon**
2	1
3	3
4	6
5	10
6	15

Tabell 1: Samanheng mellom storleik på prosjektgruppe og potensielt kommunikasjonsbehov

6.3 Kontekst 3: Politikk / verdiar / holdningar / skular

Fagfeltet programvaretesting er ikkje politikk-nøytralt. Ein kan gjerne bruka ordet 'politikk' synonymt med 'verdiar og holdningar' eller 'skular'. Orda er berre merkelappar, det viktige er det innholdet vi legg i dei.

Eit lite sitat frå boka av Hetzel [83] viser eit eksempel på kva som meinast med desse orda:

"The absence of written testing policy does not mean that no policy exists. Management always sets policies indirectly through its actions or inactions. For example, a poorly tested system implemented to meet an important deadline signals, whether they reflect true priorities or not, are quickly perceived by professional staffs and established as de facto policies. Signaling firmly to all that testing is an important concern is one of the major benefits obtained through the policy setting effort".

Det er ikkje eksklusivt for testfaget å ha ulike skular. Vi finn eit liknande forhold innanfor psykologi ("predikant"/talsperson i parentes): Strukturismen (James), behaviorismen (Watson, Skinner, Pavlov), gestaltterapien (Wertheimer), psykoanalysen (Freud, Jung), kognitivismen (Piaget) og humanismen (Rogers, Maslow).

Ulike skular innanfor programvaretesting

Innanfor testfaget omtalar Pettichord [84] fem skular. Tabellen under viser dei fem skulane samt "kortversjonen" av kva for verdiar og holdningar dei representerer i forhold til testing:

Skular innanfor programvaretesting	Verdiar og holdningar til testing
Den analytiske skulen	Ser på testing som ein rigorøs og teknisk sak. Dette synet har mange tilhengjarar innanfor universitetsmiljø.
Den standardiseringsfokuserte skulen	Ser på testing som ein måte å måla framgang på, med vekt på kostnader og gjenbrukbare standarder.
Den kvalitetsfokuserte skulen	Vektlegg prosesskvalitet som ein føresetnad for produktkvalitet. Det er viktig å holda utviklarane under kontroll, ser på si eiga rolle som "dørvakt".
Den kontekst-drivne skulen	Legg vekt på personane involvert i testinga. Søker etter feil i programmet som interessentane bryr seg om.
Den agile skulen	Har fokus på lettvektsmetodikkar (agile metodar). Legg vekt på automatisert testing. Brukar testinga til å visa at utviklinga er ferdig.

Tabell 2: Verdiar og holdningar til programvaretesting innanfor ulike skular ifølgje Pettichord

Desse 5 skulane utvikla seg i den kronologiske rekkefølgje dei står i tabellen.

Pettichord meinar skulane – eller retningane – har oppstått som eit resultat av søken etter svar på kvifor ekspertar på testing ofte er usamde. Usemje kan ikkje alltid forklarast utifrå ulik erfaringsbakgrunn; av og til må ein forstå den på bakgrunn av underliggjande ulike verdisyn.

Pettichord er ikkje åleine om å ha syn på kva ein "skule" er innanfor fagområdet programvaretesting. Jorgensen [85] skriv i det avsluttande kapittel – "Epilogue: Software Testing Excellence" – om dei lange linjer tilbake til 1978 då Myers bok – "The Art of

Software Testing" – kom. Han stiller spørsmålet om kvar programvaretesting no befinn seg langs ein tidsakse med hans eigendefinerte "skular":

Kunstart → Craftmansship → Forskning → Engineering.

Jorgensen karakteriserer si eiga bok som ein reaksjon på Myers si bok. Jorgensen oppfattar Myers å ha som holdning at det å testa eit program (eller utvikla det for den saks skuld) er ein kunstart. Undertittelen på Jorgensen si bok er "A Craftman's Approach". Som pilene over indikerer, ser Jorgensen si eiga bok å representera skulen "Craftmansship", som då må forståast å vera ein skule på eit høgare utviklingstrinn enn kunstart. Jorgensen er noko diffus i forklaringa på kva han legg i uttrykket "craftmansship". Det er tydelegvis eit personleg uttrykk for han, for han trekker linjene tilbake til sin eigen bestefar og far. Bestefaren var dansk møbelsnekkar. Faren laga verktøy og diverse utstyr. Felles for faren og bestefaren var at dei beherska det dei heldt på med, dei beherska verktøya og teknikkane, dei evna å velja rett verktøy til oppgåvene, dei hadde lang erfaring og ikkje minst yrkesstoltheit. Jorgensen er klar på at det er ikkje nokon nye "sølvkuler" som blir presentert for lesaren. Tvertimot gjev han uttrykk for at han er samd med Frederick Brooks [1] i påstanden om at det ikkje finst noko universalmiddel eller teknologi (sølvkule) for å få bukt med det fenomenet at programvareutvikling er komplisert og vanskeleg.

Kjerneverdiar for dei ulike skulane

Tabellen under viser kva for spørsmål som typisk avslører kva for verdisyn den enkelte skule har [84]:

Skular innanfor programvaretesting	**Verdiavslørande nøkkelspørsmål**
Den analytiske skulen	Kva for testteknikkar skal vi bruka?
Den standardiseringsfokuserte skulen	Korleis skal vi måla om vi gjer framgang? Når vil vi vera ferdige?
Den kvalitetsfokuserte skulen	Følgjer vi ein god prosess?
Den kontekst-drivne skulen	Kva for testing vil vera mest verdifull akkurat no?
Den agile skulen	Er dette teststeget ferdig no?

Tabell 3: Verdiavslørande nøkkelspørsmål for dei ulike skulane innanfor programvaretesting

I tabellen under er systematisert data frå Pettichord med eksempler på konkrete utsagn som reflekterer kjerneverdiar til dei ulike skulane innanfor programvaretesting:

Skular innanfor programvaretesting	**Utsegner som reflekterer kjerneverdi**
Den analytiske skulen	Programvare er eit artefakt som skal behandlast med naturvitenskapeleg logikk.
	Testing er ei særgrein av matematikken, følgjeleg er forventningen at testinga må vera objektiv, rigorøs og fullstendig.

	Testteknikkar må ha ei logisk-matematisk form ("det er eitt rett svar").
	Testing er ein teknisk operasjon.
	Testing krev ein presis og detaljert kravspesifikasjon
	Testing skal verifisera kravspesifikasjonen. "Alt anna" har ingenting med testing å gjera.
Den standardiseringsfokuserte skulen	Testinga må vera forutseibar, repeterbar og mogeleg å planleggja i forkant.
	Testinga må vera kostnadseffektiv.
	Testinga skal validera applikasjonen.
	Testinga er ein måte å måla framgangen i programvareutviklinga på.
	Aksepter ledelsens antagelsar om forhold som gjeld testing.
	Vegring mot å endra testplanar når dei er bestemt.
	Bruk V-modellen (vedlegg 1) som "samlebånd".
Den kvalitetsfokuserte skulen	Programvarekvalitet fordrar disiplin.
	Det er testinga som avgjer om utviklingsprosessen blir følgd.
	Testarane si rolle er å overvåka at utviklarane følgjer reglane.
	Testarane må beskytta brukarane mot dårleg programvare.
	Testing er ein del av organisasjonens prosessforbetringsarbeid.
Den kontekst-drivne skulen	Programvare er skapt av folk. Det er folk som har satt konteksten.
	Testinga skal finna feil. Ein feil er det som ein interessent ser på som ein feil.
	Testinga skal tilføra informasjon til prosjektet.
	Testing er ein mental aktivitet som fordrar kompetanse, såvel teoretisk som praktisk.
	Testing er ein tverrfagleg aktivitet.
	Forskning på testing er empirisk og må involvera psykologi.
	Forvent endringar i testplanar. Tilpass testplanen kontinuerleg i tråd med fortløpande testresultat.
Den agile skulen	Programvare er som ein samtale som er i gang.

	Testinga fortel oss at det aktuelle utviklingssteget er ferdig.
	Testar må automatiserast.

Tabell 4: Utsegner som reflekterer kjerneverdiar for dei ulike skulane innanfor programvaretesting.

Kvar møter vi dei ulike skulane?

Det er interessant å sjå på i kva for organisasjonar ein kan forventa å finna dei ulike skulane. Innholdet i Pettichord sin artikkel [84] er vist i tabellen under:

Skular innanfor programvaretesting	**Kan forventa å finna den praktisert hos**	**Kvar har skulen sitt opphav?**
Den analytiske skulen	Telecom	Universitet og høgskular.
	Sikkerheitskritiske prosjekt	
Den standardiseringsfokuserte skulen	Store industrikonsern.	IEEE.
	Offentlege organisasjonar.	Andre standardiseringsorganisasjonar.
		Test sertifiseringsorgan.
Den kvalitetsfokuserte skulen	Store byråkratiske prosjekt.	ISO organisasjonar.
		Software Engineering institutt.
		Kvalitetssikringsorganisasjonar.
Den kontekst-drivne skulen	Kommersiell programvareindustri.	Konferanser og workshops om programtesting.
Den agile skulen	Kommersiell programvareindustri.	Smale workshops om den aktuelle metodikken som brukast.

Tabell 5: Opphav og domene for ulike skular innanfor programvaretesting.

Kva for testteknikkar har størst appell til dei ulike skulane?

Pettichord er ganske konkret på kva for testteknikkar som har størst appell [84] til dei ulike skulane:

Skular innanfor programvaretesting	**Testteknikk med størst appell**
Den analytiske skulen	Dekningsgrad for testtilfelle.
Den standardiseringsfokuserte skulen	Testmatriser for krav i kravspesifikasjonen (sporing av krava i testinga).

Den kvalitetsfokuserte skulen	Den formalistiske dørvaktrollen ("denne programvara er ikkje ferdig før enn testarane seier at den er ferdig").
Den kontekst-drivne skulen	Utforskande testing (design av testar og eksekvering av desse testane skjer parallellt. Rask læring).
Den agile skulen	Einheitstestar (som brukast til test-driven utvikling).

Tabell 6: Testteknikkar med størst appell for dei ulike skulane for programvaretesting

Styrke og svakheit til dei ulike skulane

Dei ulike skulane har ulik styrke og svakheit stilt overfor ulike utfordringar som testing kan by på. Pettichord [84] har vurdert kor godt dei vil vera eigna stilt overfor to konkrete utfordringar:

- Kravspesifikasjonen manglar eller er dårleg
- Testaren maglar sertifisering som testar

Viss kravspesifikasjonen er dårleg, vurderer Pettichord den kontekst-drivne skulen og den agile skulen som best eigna for situasjonen. Den utforskande naturen til den kontekst-drivne skulen vil greia seg bra. Den agile skulen har som holdning at konversasjon er viktigare uansett enn dokumentasjon. For den analytiske skulen, som legg stor vekt på at testinga skal verifisera om programmet er i samsvar med kravspesifikasjonen, og som ser "alt anna" som testinga uvedkommande, vil ikkje dette vera den beste situasjonen. For den kvalitetsfokuserte skulen vil testarane mangla ein kravspesifikasjon å navigera etter når dei skal "styra" utviklarane, og det kan bli problematisk.

Viss testarane manglar sertifisering som testarar, vil ein i den kontekst-drivne skulen sjå på reell kompetanse som viktigare enn formell kompetanse. I den analytiske skulen vil ein uansett sjå meir etter formell universitetsutdanning enn etter sertifiseringar. Dei som blir mest "hjelpelause" i ein slik situasjon er dei som sterkast ser sertifisering som uttrykk for status, nemleg den standardfokuserte- og den kvalitetsfokuserte skulen.

6.4 Kontekst 4: Type applikasjon

Jorgensen [85] presenterer ei topp-10 liste over beste praksisar basert på sine eigne praktiske erfaringar. Han framhevar at kven ein bør velja i miksen av dei 10 beste praksisar, avheng av det aktuelle prosjektet, som i sin tur avheng av naturen til den aktuelle applikasjonen som skal testast. Han deler applikasjonen sin type inn i 3 grupper:

- **Systemkritiske**. Dette er typisk applikasjonar som brukast i romfart, sjukehus og forsvar, der felles for desse er store krav til presisjon og pålitelegheit.
- **Tidskritiske**. Dette er applikasjonar der det er viktig med rask utvikling for å komma først på marknaden med eit produkt for å posisjonera seg og få høg markedsandel.

- **Vedlikehold av gammal kode**. Dette er den vanlegaste form for utvikling. Ifølgje Jorgensen representerer vedlikehold av kode typisk 75% av all programmeringsaktivitet i ein organisasjon. Ofte blir slikt vedlikehold utført av ein annan person enn den som laga koden.

Tabellen under viser hans 10 beste praksisar, og kven av desse han meinar passar best for tre ulike typar applikasjon:

Beste praksis	**Systemkritiske**	**Tidskritiske**	**Vedlikehold av gammal kode**
Modell-driven utvikling	X		
Identifikasjon av testnivå	X	X	X
Modell-basert systemnivå testing	X		
Grundig systemtesting	X		
Incidensmatriser til hjelp for regresjonstesting	X		X
Bruk av MM-stier for integrasjonstesting	X		
Kløktig kombinasjon av spesifikasjonsbasert og kodebasert einheitstesting	X		X
Måling av dekningsgrad basert på karakteristika ved individuelle einheiter	X		
Eksplorativ testing			X
Test-driven utvikling		X	

Tabell 7: Ti beste praksisar av testteknikkar for ulike applikasjonstypar ifølgje Jorgensen

6.5 Korleis bør Arkade testast i lys av dette kapittel?

Konteksten forskningsparadigme er relevant sett ut ifrå eit metodisk synspunkt, men det treng nødvendigvis ikkje vera så relevant ut ifrå eit praktisk synspunkt.

Konteksten organisering og leiing av testarbeidet er ikkje så relevant for Arkade, sidan denne masteroppgåva blir utført av meg åleine.

Konteksten skule er relevant fordi bedriftskultur og den rolle testinga har i ein organisasjon gjerne kan kjennast att i ulike verdisyn ("slik gjer vi det her"). Status for kravspesifikasjonen til Arkade og min kompetanse dreg i retning mot den kontekst-drivne skulen eller den agile skulen.

Applikasjonen Arkade sin type plasserer den i kategorien 'Vedlikehold av gammal kode'.

Dei 4 av dei 10 beste praksisar som ifølgje Jorgensen då gjeld er:

- Identifikasjon av testnivå.
- Incidensmatriser til hjelp for regresjonstesting.
- Kløktig kombinasjon av spesifikasjonsbasert og kodebasert einheitstesting
- Eksplorativ testing

Identifikasjon av testnivå

Jorgensen skriv at [85]:

"Any application (unless it is quite small) should have at least two levels of testing – unit and system. Larger applications generally do well to add integration testing. Controlling the testing at these two levels is critical. Each level has clearly defined objectives, and these should be observed. System-level test cases that exercise unit-level considerations are both absurd and a waste of precious test time".

Termen "testnivå" (vedlegg 1) er tradisjonelt mest brukt viss ein følgjer V-modellen (vedlegg 1) der konteksten er eit systemutviklingsprosjekt og der utvikling og testing gjerne skjer parallellt. Konteksten for masteroppgåva er å testa/feilsøka ein applikasjon der utviklingsprosjektet skjedde for ein del år tilbake. Det er likevel relevant å bruka termene einheitstesting og systemtesting for arbeidet med masteroppgåva. Dette viser at termene einheitstesting og systemtesting blir brukt både i konteksten "testnivå" og i konteksten "testmetode". Jorgensen refererer til einheitstesting og systemtesting i testnivå-kontekst. For masteroppgåva er det testteknikk-kontekst som er relevant.

Incidensmatriser til hjelp for regresjonstesting

Om dette skriv Jorgensen [85]:

"Both traditional and object-oriented software projects benefit from an incidence matrix. For procedural software, the incidence between mainline functions (sometimes called features) and the implementing procedures is recorded in the matrix. Thus, for a particular function, the set of procedures needed to support that function is readily identified. Similarly for object-oriented software, the incidenence between use cases and classes is recorded. In either paradigm, this information can be used to:

- *Determine the order and contents of builds (or increments)*
- *Facilitate fault isolation when faults are revealed (or reported)*
- *Guide regression testing"*

Regresjonstesting [66] tyder gjentaking av ein eller fleire testar eller testfasar for å verifisera at feilrettingar eller endringar ikkje har introdusert nye feil eller uønska effekter, og at systemet fremleis tilfredsstiller spesifiserte krav.

Dette tyder at ein held styr på kva for feilrettingar som kan få konsekvensar for andre deler av programmet.

Eg veljer bort incidensmatriser ved testing av Arkade, fordi eg ikkje kjenner designet av systemet godt nok frå utviklingsfasen. Men regresjonstesting er eit råd eg tek med meg.

Kløktig kombinasjon av spesifikasjonsbasert og kodebasert einheitstesting

Sitat frå boka til Jorgensen [85]:

"Neither specification-based nor code-based unit testing is sufficient by itself, but the combination is highly desirable. The best practice is to choose a specification-based technique based on the nature of the unit (see Chapter 8), run the test cases with a tool to show test coverage, and then use the coverage report to reduce redundant test cases and add additional test cases mandated by coverage".

Eg meinar dette ikkje passar heilt inn i konteksten med testing av Arkade. Fordi eg har ikkje vore med på utviklinga som utviklar og kjenner ikkje koden og designet godt nok. Eg kunne vorte kjent med det, men det ville gjeve masteroppgåva eit anna fokus enn eg hadde tenkt. Ein annan grunn er at Arkade ikkje vart utvikla på grunnlag av nokon klar og tydeleg kravspesifikasjon.

Eksplorativ testing (utforskande testing)

Noko som Jorgensen skriv på side 407 [85] passar svært godt for testing av Arkade:

"Exploratory testing is a powerful approach when testing code written by someone other than the tester. This is particularly true for maintenance on legacy code.".

I forhold til Arkade kan vi tolka "legacy code" [86] som kode som ikkje lenger blir støtta og vedlikeholde av ein leverandør.

Det foreligg ein dårleg kravspesifikasjon for programmet, den vart i stor grad til mens programmet vart utvikla. Eg er heller ikkje i utgangspunktet kjent med kjeldekoden fordi eg ikkje var involvert i utviklinga av programmet i ein slik rolle. Eg føler at både den kontekst-drivne skulen og den agile skulen har bra tilnærming til denne situasjonen.

Kva ER "Beste Praksis"?

Uttrykket "Beste Praksis" er mykje brukt i mange samanhengar, også i programvareindustrien.

Denne evalueringa av "Beste Praksisar" for Arkade viser at 3 av desse 4 ikkje blir oppfatta av meg som sådan. Dette kan gje grunn til å dvela ved bruken av uttrykket i litteraturen og kor allmenngyldig det er.

Generelt fører uttrykket med seg mykje fokus på metode og teknikk, mindre på kompetanse. Bach og Schroeder [87] uttrykker skepsis til dette:

"Because test techniques are heuristic, focus not on the technique, but your skills as a tester. Develop your skills."

Vidare skriv dei:

"We recommend that you refuse to follow any technique that you don't yet understand, except as an experiment to further your education. You cannot be a responsible tester and at the same time do a technique just because some perceived authority says you should, unless that authority is personally taking responsibility for the quality of your work. Instead, relentlessly question the justification for using each and any test technique, and try to imagine how that technique can fail."

Whitmill [88] skriv i ein interessant artikkel om råd han gjev at "These suggestions are based on my observations of people I respect for their test expertice". Eit av råda er:

"Better Testing is Context-Driven: Better testing requires thinking. You must base your testing on what is best in your context, not simply repeating a previous test or blindly following a "best practice". A "best practice" in one context may be a "worst practice" in another!"

Her får Whitmill støtte av Roden [63] som tykkjer at uttrykket "Beste praksis" gjev feil assosiasjonar. For ingenting er ifølgje han best i alle situasjonar. Det kjem mellom anna an på kva for kompetansenivå ein er på. For personar som er profesjonelle eller ekspertar, vil "Beste praksis" vera i vegen viss det er å forstå som noko som skal etterlevast konsekvent. Det vil føra til at flinke folk blir frustrert og det vil ha ein konserverande effekt i eit miljø som tvert imot bør vera kreativt. Ekspertar brukar, og bør bruka, sitt instinkt, ifølgje Roden.

7 Utforskande testing

På bakgrunn av tidlegare kapittel i denne masteroppgåva, er det sterke indikasjonar på at testteknikken utforskande testing bør vera ein av testteknikkane når Arkade skal testast. Eksempler på slike indikasjonar er:

- I kapittel 3 uttalar testar A under overskrifta 'Testmetoden i seg sjølv' at han har erfart at vanleg testing og utforskande testing er komplementære.
- I kapittel 4 visest ein tabell presentert av Julie Gardiner der ho plasserer testteknikken 'Exploratory Testing' i øvre høgre kvadrant (testteknikkar med evne til å finna feil som med stor sannsynlegheit vil inntreffa og som tyder mykje for kvaliteten av programmet).
- I kapittel 6 blir det dokumentert at denne testteknikken er aktuell når det gjeld vedlikehold av gammal kode, når det er dårleg kjent om krava er implementert i programkoden eller når rask læring er nødvendig. Alle desse forholda stemmer for den konteksten Arkade blir testa i.
- Det er ikkje utvikla noko teknisk testmiljø (vedlegg 1) for Arkade. Det er grunn til å tru at dette vil krevja kreativitet, ein eigenskap som utforskande testing har ord på seg for å gje rom for. Verdien av eit testmiljø er godt dokumentert i kapittel 4 i masteroppgåva, jamfør erfaringar frå Julie Gardiner, Lloyd Roden, Telenor, Storebrand og Statens Pensjonskasse.

7.1 Kva er utforskande testing?

Den definisjonen som har utkrystallisert seg, er [89]:

> *"Exploratory testing is simultaneous learning, test design and test execution".*

James Bach ser til ei viss grad all testing som utforskande testing. Denne testmetoden er enkel og lite påakta. Bach beskriv situasjonen slik [89]:

> *"So it is with exploratory testing (ET): simultaneous learning, test design and test execution.This is a simple concept. But the fact that it can be described in a sentence can make it seem like something not worth describing. Its highly situational structure can make it seem, to the casual observer, that it has no structure at all. That's why textbooks on software testing, with few exceptions, either don't discuss exploratory testing, or discuss it only to dismiss it as an unworthy practice."*

Testteknikken er også forklart i vedelgg 1 og på Wikipedia [27].

Bach beskriv 5 grunnelement i utforskande testing:

- Test design: Ein utforskande testar er først og fremst ein testdesigner. Alle kan designa testar tilfeldig, den som er flink til det gjer det på ein måte som systematisk utforskar

produktet. Dette krev ferdigheiter som å analysera produktet, bruka verktøy og tenka kritisk.

- Nøyaktig observasjon: Ein som testar etter eit script, forheld seg mest berre til det som scriptet rapporterer. Ein utforskande testar derimot bør leita etter eitkvart teikn til noko uvanleg.

- Kritisk tenking: Ein utforskande testar må leita etter feil såvel i sin eigen tankegang som i programmet.

- Kreative idear: Ein utforskande testar kan bruka heuristikker (vedlegg 1, eller sjå [90]), men ikkje sjå seg blind på dei. Ein heuristikk er ein statisk testteknikk for å testa brukbarheit ved å undersøkja samsvar mellom eit brukargrensesnitt og anerkjente prinsipp for brukbarheit.

- Hjelperessursar: Ein god utforskande testar byggjer faglege nettverk og erfaringsdatabasar av testdata, og ser etter høve til å bruka desse under testing.

Den kontekst-drivne skulen ser på utforskande testing som ein mental aktivitet, og Bach underbyggjer dette synet ved å skapa ein analogi mellom utforskande testing og sjakkspeling. Prosedyrer er av underordna interesse i sjakk, det er kompetansen som tel. Det som avgjer kampen er dei val spelaren gjer på alle stadium i spelet, og som er basert på eit samansett konglomerat av erfaringar [89]:

"A basic strategy of ET is to have a general plan of attack, but allow yourself to deviate from it for short periods of time. Cen Kaner calls this the "tour bus" principle. Every people on a tour bus get to step off it occasionally and wander around. The key is not to miss the tour entirely, nor to fall asleep on the bus".

Kjernespørsmålet i den kontekst-drivne skulen er: Kva for testing vil vera mest verdifull akkurat no? Bach underbyggjer også dette synet ved å skapa ein analogi mellom å bruka utforskande testing og setja saman puslespel: Du tek opp ein bit frå puslespelet og studerer den. Kvart blikk på biten kan samanliknast med eit testtilfelle ("Kvar passar denne biten inn i samanhengen?"). Ein kan velja å setja saman puslespelet meir systematisk, for eksempel ved å starta med hjørna først, deretter kantane. Ein kan stilla seg spørsmålet: Er det riktig å gjera dette på ein rigorøs måte? Forfattaren ber oss sjå føre oss korleis det vil fungera i praksis om vi skulle dokumentera alle testtilfella (puslespelbrikkene) på førehand. Det ville bli ganske håplaust, og resultatet ville neppe stått i forhold til innsatsen. For ein ville kanskje ikkje vita korleis bildet såg ut før ein begynnte på puslespelet. Vi måtte kanskje sjå det føre oss iallfall, sjølv om analogien til puslespel sviktar litt der, sidan resultatet for puslespel ofte er eit visuelt bilete utanpå loket på øskja med brikkene inni, mens programvare er usynleg i sin karakter.

Bach skriv at mens han set saman eit puslespel, forandrar han heile tida framgangsmåten etter kvart som han lærer meir om spelet. Viss han for eksempel kjem til eit parti med ein spesiell farge, så kan han leggja brettet til sides og heller samla for seg sjølv alle brikkene han kjem over som har den fargen. Når han driv med testing, så kan han plutseleg endra frå ein type testing til ein annan, fordi han går lei, og treng å klara opp hjernen. Av og til kan han føla at det han held på med er altfor dårleg organisert, og då kan han ta eit steg tilbake, søka overblikk, analysera situasjonen, og forsøka seg med ein ny angrepsvinkel eller tilnærming. Likevel ser han det slik at det er ein flyt i arbeidet, det er kontinuitet til eikvar tid. Er det ikkje

akkurat slik situasjonen er når ein skal setja saman eit puslespel også? spør han. Og han gjev svaret:

"This truth is at the heart of any exploratory investigation, be it for testing, development, or even scientific research or detective work.".

Bach ser fordeler ved å bruka utforskande testing framfor testscipt [89]:

"The power of explorarory tests can be optimized throughout the test process, whereas scripts, because they don't change, tend to become less powerful over time. They fade for many reasons, but the major reason is that once you've executed a scripted test one time and not found a problem, the chance that you will find a problem on the second execution of the script is, in most circumstances, substancially lower than if you ran a new test instead.".

Ifølgje Bach er det ingen studiar som samanliknar utforskande testing med scripttesting:

"There are no reasonable numbers; no valid studies of testing productivity that compares ET with scripted testing. All we have are anecdotes.".

7.2 Når passar det å bruka utforskande testing?

Bach meinar at det generelt vil passa å bruka utforskande testing i einkvar situasjon der det ikkje er opplagt kva den neste testen bør vera, eller der testinga ikkje er fullstendig diktert på førehand. For eksempel i følgjande situasjonar [89]:

- Du treng raskt svar når det gjeld eit nytt produkt eller ein ny eigenskap.
- Du treng å læra programmet raskt.
- Du har allereide testa programmet ved bruk av scripter, og søker å komplettera testinga.
- Du ønskjer å finna den viktigaste feilen på kortast mogeleg tid.
- Du ønskjer å sjekka arbeidet til nokon andre ved å gjera ei kort, uavhengig undersøking.
- Du ønsker å undersøkja og isolera ein spesiell defekt.
- Du ønskjer å undersøkja status for ein spesiell risiko, for å evaluera behovet for scripttesting på eit område.

Bach sine analogiar mellom utforskande testing og ein fotballkamp, eit sjakkparti eller å leggja eit puslespel, brukar ei referanseramme som mange, inkludert meg sjølv, kjenner seg att i. Mi subjektive oppfatning er at dette går til kjernen av kva all testing innerst inne dreiar seg om, nemleg kognitiv læring og adferd.

Korleis blir arbeidet med feilsøking og feilretting i SAS-program oppfatta å vera? Vil det vera moment her som er analoge til slik Bach beskriv utforskande testing? Utforskande testing søker alternativ viss noko ikkje fører fram. Eg finn den holdning Delorio her gjev uttrykk for, å passa godt til utforskande testing [82]:

"If you are stuck, and you have the luxury of time, use it to your advantage. Leave the problem program and work on something else, even for a short while. It's remarkable how often even experienced programmers get stuck in one path of attack on the problem. Coming back to the program with a fresh attitude clears the debris from the old path and often identifies alternate solutions. Failing that, get a colleague's fresh pair of eyes to examine the problem – it's striking how many times someone will immediately locate the missing semicolon that eluded you for hours. Don't rush the process. Believe, as Gerald Weinberg says, that you should "never debug standing up." If you don't have the luxury of a break, at least proceed a bit more deliberately. The write-run-review cycle in most program development environments encourages rapid resubmission of programs. Very often, this simply means that a panicky programmer at loose ends tries one rapid-fire solution after another without really examining the underlying problem. Slow down, spend some time away from the keyboard, and fully examine the Log and any other outputs."

Det å gjera erfaringar undervegs, og ta hensyn til desse i problemløysinga vidare, er ein vesentleg del av utforskande testing. Eg oppfattar at Delorio gjev uttrykk for ei tilnærming som passar godt inn med utforskande testing her [82]:

"Once debugging is complete, reflect on what the correctly coded program looks like. Also consider how you got there. How was the problem identified (syntax or logic)? How did the problem-solving process proceed? Did you rely on experience, intuition, documentation, or other resources? Which were most effective? What types of false leads did you encounter? Would different logic, design, or program presentation make similar problems less frequent in the future? Make notes about the process by using comments in the program or keeping a written log containing both the problem description and its solution. Even if you think you'll never forget the road to the solution, you probably will. Documentation will deal of head scratching later on. "

Den dokumentasjon av problemløysing Delorio her snakkar om, oppfattar eg som nødvendig, men likevel minimalistisk. Kommentarar i kjeldekoden prioriterast framfor lange tekstdokument.

Utforskande testing legg stor vekt på forståing av konteksten feil opptrer i. Det oppfattar eg at Delorio også gjer [82]:

"Successful debugging means addressing the bug's source, not just its current manifestation. A data source, for example, may have unexpected values. The solution may be to insert an IF statement whose condition sets the bad values to missing. This is a stopgap measure at best. It does not address the reason why the values occurred and whether they are legitimate. It fairly guarantees that another bug will be detected later, identifying yet

more unexpected values. Take the time to understand and deal with the reason the data were unanticipated, and not simply code around the problem. "

7.3 Kritikk av utforskande testing

Zylberman [91] skiljer mellom ad hoc testing og utforskande testing. Med ad hoc testing forstår han testing utan at det foreligg nokon testplan i det heile. Bach [92] imøtegår myten om at utforskande testing ikkje kan kallast planmessig testing. Bach beskriv testplanen som "the set of ideas that guide the test process". Ifølgje Bach er det logisk at testing skjer forut for det å laga ein testplan. Han dreg analogien til det å avholda ein pianokonsert. Ein slik konsert er planlagt, men forut for ein detaljert plan for gjennomføringa av konserten, var det "*some kind of composing and practicing process*". Men denne forutgåande testinga var ikkje ledsaga av nokon rigorøs testplan. Før ein kan planleggja noko, meinar Bach at ein må testa for å finna ut kva ein skal testa. For dei som set utforskande testing i kontrast til planmessig testing, skriv Bach:

"Exploratory testing is exactly the kind of thing that comes before rigor and makes rigor possible".

Han forsvarar også den korte planlegginshorisonten som utforskande testing har. Han dreg ein analogi til fotball:

"In football, when a quarterback finds his receivers are too well covered, he may quickly make a new plan to run with the ball himself. An instant later, he executes his plan. Thinking fast is no crime against plankind".

I artikkelen av Agruss og Johnson [93] ser forfattarane på ad hoc testing og set slik testing i perspektiv saman med andre former for utforskande testing. Dette tyder på at dei ser på utforskande testing som ei tilnærming heller enn ein formelt definert testteknikk. Ad hoc testing definerast som "an exploratory case that you expect to run only once". Denne teknikken blir satt i relieff til regresjonstesting. Desse to testteknikkane blir satt i bås med henholdsvis formelle testar som regresjonstesting, systemtesting, og akseptansetesting på den eine sida og mindre formelle testar som utforskande testing, ad hoc testing, alfatesting og betatesting (vedlegg 1, eller sjå [94]) på den andre sida. Artikkelen sitt poeng er at mindre formelle testar har sin styrke i å finna feil, mens formelle testar har ein større styrke i det å byggja tillit til programvara hos testaren og brukaren.

Bach, Bach og Bolton [95] støttar også synet på å setja utforskande testing i kontrast til testing ved bruk av script. Men dei ser ikkje dette som testteknikkar, men som test-tilnærmingar. Fordi alle testteknikkar kan utførast enten i form av utforskande testing eller ved hjelp av eit testscript.

Om ein ser på utforskande testing som ei tilnærming, ikkje ein strikt definert teknikk, kva så? Eller spurt på ein annan måte: Er nokon testteknikkar i det heile strikt definerte når det kjem til stykket? Kritikken om at utforskande testing er vag og ullen i definisjoen, kan like gjerne rettast mot mange av dei termane som blir brukte innan fagområdet testing av programvare. Personleg trur eg ikkje at termar vart oppfunne for å stå i vegen for oss, men for å vera til hjelp.

Uansett, det kan vera greit å ta med seg eit velmeint råd frå Bach, der han minner om at ein får ikkje svart belte verken i karate eller programvaretesting berre av å lesa bøker:

”Exploratory testing can be described as a martial art of the mind. It’s how you deal with a product that jumps out from the bushes and challenges you to a duel of testing. Well, you don’t become a black belt by reading books. You have to work at it. Happy practicing.”

8 Testing av Arkade

Siste del av kapittel 6 var dedikert til spørsmålet om korleis Arkade bør testast. Etter å ha lese kapittel 7 har vi litt betre føresetnader for å gje svar på det spørsmålet. Dette kapittelet er den empiriske delen av masteroppgåva. Masteroppgåva no er kommen dit kor eg forsøker å gje svar på dei spesielle problemstillingane:

- *Korleis bør Arkivverket sitt SAS-program Arkade testast?*
- *Kor lett testbart er dette programmet?*
- *Korleis kan det eventuelt gjerast meir testbart?*

På bakgrunn av kapittel 6 og 7, er det to holdepunkt som står klart for meg for korleis eg vil testa Arkade:

- Testteknikken utforskande testing (kapittel 6 og 7).
- Testing ved bruk av eigenutvikla testapplikasjon. I praksis å etablera eit testmiljø (vedlegg 1) for Arkade (innleiinga til kapittel 7).

Eg veljer difor å først gå nærmare inn på desse to kulepunkta først i dette kapittelet, og i denne rekkefølgja, sidan utforskande testing har vore ein vesentleg føresetnad og hjelp for å utvikla ein testapplikasjon.

8.1 Testteknikken utforskande testing

I innleiinga til kapittel 7 om utforskande testing er det nemnd nokre sterke indikasjonar på kvifor denne testteknikken bør veljast. Det gjeld praktisk erfaring både frå testar A og Julie Gardiner.

I kapittel 6 (punkt 6.5) refererast til eit råd frå Jorgensen:

”Exploratory testing is a powerful approach when testing code written by someone other than the tester. This is particularly true for maintenance on legacy code.”.

Denne beskrivinga passar svært godt på konteksten for testinga av Arkade i dette masterprosjektet.

Utforskande testing har ein iterativ karakter. Dette kapittelet har vorte til som eit resultat av ein iterativ prosess mellom litteraturstudiar og praktisk kodeutvikling. *Følgjeleg er dette eit kapittel som både reflekterer teori og praktisk testarbeid*:

- Litteraturstudiene har hatt fokus mot programmeringsspråket SAS, og eg har hatt svært god nytte av brukargrupper på nettet. Det er ei god oversikt over brukargrupper globalt for SAS [96]. Mellom anna SESUG (SouthEast SAS Users Group) [97] og NESUG (NorthEast SAS Users Group) [98].

- Den praktiske kodeutviklinga og testinga er plassert i eigne vedlegg. Rapporten trekker fram kort kva som er gjort og resultat av arbeidet, men forklaringa av korleis ein har komme fram til resultata, er så nært knytt til den praktiske kodeutviklinga og kommentarane til programkoden at dette stoffet er reservert for vedlegga. Det blir referert til vedlegga i rapporten.

Utforskande testing søker å bruka testteknikkar som passar til ulike formål. I mange situasjonar har eg har valt statisk testing [75]:

"Static testing is a form of software testing where the software isn't actually used. This is in contrast to dynamic testing. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code and/or manually reviewing the code or document to find errors. This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used. From the black box testing point of view, static testing involves reviewing requirements and specifications. This is done with an eye toward completeness or appropriateness for the task at hand. This is the verification portion of Verification and Validation".

Eksempler på bruk av statisk testing i dette kapittelet er:

- Testing av struktur og vedlikeholdbarheit (vedlegg 1, sjå også [99]) i koden.
- Testing av gyldigheitsområde for markovariablar.
- Testing av eksekverbar kode frå ein makro ved hjelp av ein annan makro.
- Testing av korleis Arkade tek vare på evidens til bruk for å bli testa.
- Testing av bruken av makroparametrar i Arkade.
- Testing som evaluerer om Arkade brukar innebygd funksjonalitet i SAS for å testa programmet.

I andre situasjonar har eg sett det som meir hensiktsmessig å bruka dynamisk testing [100]:

"Dynamic testing (or dynamic analysis) is a term used in software engineering to describe the testing of the dynamic behavior of code. That is, dynamic analysis refers to the examination of the physical response from the system to variables that are not constant and change with time. In dynamic testing the software must actually be compiled and run; Actually Dynamic Testing involves working with the software, giving input values and checking if the output is as expected. These are the Validation activities. Unit Tests, Integration Tests, System Tests and Acceptance Tests are few of the Dynamic Testing methodologies. Dynamic testing means testing based on specific test cases by execution of the test object or running programs".

Eksempler på testteknikkar i dette kapittel som føreset dynamisk testing er:

- Testing av returkode.

- Testing ved bruk av kode som avsluttar eksekveringa.
- Testing ved å la globale makrovariablar erstatta kommentar-teikn.
- Testing som dreiar seg om parsing av SAS-loggen.
- Testing av kombinasjonar av inndata i metadatafila.

Testteknikken utforskande testing kan kanskje verka noko usynleg i seg sjølv i dette kapittelet, men ein skal då vita at den har vore "piloten" som har gjort val heile tida og som har hatt som leietråd: Kva er den beste testteknikken akkurat no?

8.2 Testing ved bruk av eigenutvikla testapplikasjon

Testteknikken som ligg til grunn for etablering av testmiljø for Arkade er utforskande testing. I innleiinga til kapittel 7 om utforskande testing er det nemnd nokre sterke indikasjonar på kvifor denne testteknikken bør veljast. Eg skriv der at det er grunn til å tru at utvikling av eit testmiljø for Arkade vil krevja kreativitet, noko som utforskande testing har stort rom for.

Bentley [9] definerer testmiljø som "The technical environment, data, work area, and interfaces used in testing". For testinga av Arkade har testmiljø og den eigenutvikla testapplikasjonen vore synonyme.

Verdien av eit testmiljø er godt dokumentert så langt i masteroppgåva, både av Julie Gardiner, Lloyd Roden, Telenor, Storebrand og Statens Pensjonskasse.

Eg har valt SAS som programplattform for å utvikla ein testapplikasjon for Arkade. Det står for meg som rasjonelt å bruka det same programmeringsspråket som Arkade er utvikla i. Feilsøking og retting av feil i koden føreset at ein forstår SAS godt nok til å gjera det. Likeeins om det blir avdekka forhold som føreset vidare utvikling av eksisterande funksjonalitet. Litteraturstudiar viser ein rikholdig litteratur på SAS-relevante teknikkar for testing og debugging, noko referansar brukt utover i dette kapittel vil visa. Dessutan har eg ein del erfaring med SAS som programmeringsspråk. Ifølgje teorien om utforskande testing (kapittel 7) vil slik kompetanse (både utvikling og praktisk testing av eigenutvikla kode) vera ein stor fordel for å ha nytte av testteknikken utforskande testing.

Det har vore mykje prøving og feiling, mange iterasjonar med ulike val av design for testprogrammet. Kva for makroar som skulle utviklast i SAS for å vera fundament i eit testmiljø for Arkade. Desse programfilene og makroane er ein vestentleg del av testmiljøet for å testa Arkade.

I det følgjande blir det gjort greie for val for testmiljøet og desse blir grunngjevne. I tillegg er ein del stoff trekt ut frå hovudrapporten og plassert i vedlegg 3. Det gjeld dokumentasjonen av:

- Styrefila for testapplikasjonen Arkade_TestApp.
- Programfila for å køyra heile testapplikasjonen (integrasjonstesting).

- Ein del av dei viktigaste makroane utvikla for å etablera testmiljøet.
- Ei dedikert programfil per testtilfelle.

Grunngjeving for diverse val for testmiljøet

Etablering av testmiljøet representerer ei rekke val, som her blir presenterte saman med grunngjeving for desse vala. Vala for design av testmiljøet er:

- testtilfelle er rammeverk for den dynamiske testinga.
- skilje mellom produksjonsmiljø og testmiljø.
- bruk av enkel og hensiktsmessig namnekonvensjon.
- dedikert Autoexec.sas-fil for testapplikasjonen.
- eitt bibliotek per testtilfelle.
- enkelt vedlikehold ved endring av katalogstruktur.
- katalogstruktur med to nivå.
- enkelt vedlikehold ved endring av filnamn.
- to modi for kodeeksekvering; feilsøking og produksjon.
- sentralisert vedlikehold og eksplisitt kompilering av koden.
- %INCLUDE som valt makrobibliotek.
- kvar makro får si eiga programfil.

I det følgjande vil eg gå litt nærmare inn på desse kulepunkta:

Testtilfelle er rammeverk for den dynamiske testinga

Testtilfelle er ein generell term, og seier ikkje noko om kva for testteknikk som blir brukt (vedlegg 1, sjå også [37]). Det seier berre at ”her blir eitt eller anna fenomen testa på ein eller annan måte”. Som rammeverk å rekna, er nytten av testtilfelle på dette stadiet i utforminga av testmiljøet mest nyttig som organisatorisk ramme for inndeling/oppdeling i dei fenomener ein ønskjer å testa.

Dei fleste testtilfelle vil truleg bestå av ulike inndata for både datafil og metadatafil.

Skilje mellom produksjonsmiljø og testmiljø

I industrien stressar ein gjerne at produksjonsmiljøet og testmiljøet ikkje eingong skal liggja på den same serveren. Så nøye har ikkje eg teke det. Eg har late tenkt både Arkade og testapplikasjonen liggja på den same stasjonen (D-stasjonen) på maskina mi, men på ulike katalogar. Testmiljøet er ein applikasjon i seg sjølv, som fungerer uavhengig av Arkade-applikasjonen. Enkelte feil i Arkade kan det vera risikofritt å endra, mens for andre endringar kan det vera uvisse om korvidt det vil påverka seinare deler av programmet, eller uvisse om

endringa innfører utilsikta feil. Difor er det ein kopi av den originale Arkade-applikasjonen eg har brukt.

Bruk av enkel og hensiktsmessig namnekonvensjon

Eg tykkjer følgjande namnekonvensjon er hensiktsmessig for programfiler som skal eksekvera eit testtilfelle: <Filindikator for at det er eit testtilfelle>_<Testtilfelle nr>.

Eksempel: T_01.sas.

Dedikert AUTOEXEC-fil for testapplikasjonen

Eg har erfart at det er vanleg i industi-samanheng at ulike prosjekt har si eiga AUTOEXEC.sas-fil. Det kan vera parametrar så som kvar filer skal lesast ifrå og kvar dei skal skrivast til som er spesifikke for kvart prosjekt. Det har praktiske fordeler at testapplikasjonen blir mest mogeleg uavhengig av Arkade-applikasjonen.

Eitt bibliotek per testtilfelle

Med bibliotek meinast i denne samanheng SAS-bibliotek i SAS Environment. Dette er ein logisk peikar til ein katalog på harddisken der filer som visuelt presenterer seg som datasett i SAS, er plasserte. Eg vel eitt bibliotek for kvart testtilfelle. Det kan for eksempel vera aktuelt å testa det same fenomenet med fleire enn eitt testtilfelle. Viss ein brukar ulike bibliotek, kan datasetta behalda det namnet dei har i Arkade-applikasjonen. Å måtta gå inn i koden og endra namn på datasetta frå testtilfelle til testtilfelle, var for tungvint, og uaktuelt.

Enkelt vedlikehold ved endring av katalogstruktur

Dette er i tråd med prinsippet om design for vekst og utvikling. Det er eit sunt og generelt prinsipp i systemutvikling.

Katalogar blir dynamisk referert ved hjelp av globale makrovariablar. Viss ein katalogsti eller eit katalognamn blir endra, skjer oppdateringa på kun ein stad, der makrovariabelen er definert. Alternativet ville vore å retta den på kanskje hundre stadar i programkoden.

Katalogstruktur med to nivå

Katalogstrukturen i testapplikasjonen er på to nivå. I prinsippet kan ein plassera kodefiler kvar ein vil. Poenget med to nivå er at ting som synest logisk å høyra saman, er plassert på same nivå. For utdata blir det oppretta ein underkatalog for kvart testtilfelle.

To modi for kodeeksekvering; feilsøking og produksjon

Hensikten er at det kan vera ulike behov til ulike tider for kor mykje informasjon eg ønskjer å få frå SAS-loggen. SAS sin systemdokumentasjon inneheld ei forklaring på kva SAS-loggen er [101]:

"The SAS log is a record of everything that you do in your SAS session or with your SAS program. Original program statements are identified by line numbers. Interspersed with SAS statements are messages from SAS. These messages might begin with the words NOTE, INFO, WARNING, ERROR, or an error number, and they might refer to a SAS statement by its line number in the log".

Til tider kan overflod av informasjon frå loggen vera ei ulempe, fordi det faktisk kan auka sannsynlegheiten for at feilmeldingar ikkje blir oppdaga. På den andre sida, når ein feil først er oppdaga, kan det vera fordelaktig å enkelt kunna endra innstillingar slik at loggen gjev meir detaljert informasjon.

Sentralisert vedlikehold og eksplisitt kompilering av koden

For å forenkla vedlikehold og utvikling av testapplikasjonen, vil store deler av vedlikeholdet skje i tre filkatalogar:

- Kode_for_autoexec
- Kode_for_sentralkoden
- Metadata

Alle makroar er lagra i katalogane for dei to første kulepunkta. Desse katalogane inneheld henholdsvis programfilene MAKROER_AUTOEXEC_ARKADE_TESTAPP.sas og MAKROER_SENTRALKODEN_ARKADE_TESTAPP.sas. Desse programfilene ligg saman med programfiler som er reine definisjonsfiler for kvar sin makro, og avgjer eksplisitt kven av makroane som skal kompilerast. Ein makro kan såleis godt liggja "ubrukt" i desse katalogane. Eller katalogane kan vera eit "reservoar" for fleire applikasjonar.

%INCLUDE som valt makrobibliotek

Det er 3 måtar å velja makrobibliotek på [17]:

"There are three types of macro libraries; %INCLUDE, Compiled Stored Macros, AUTOCALL Macros. Each has its advantages and disadvantages, and best of all they can be used in conjunction with each other!"

Carpenter [17] kan kanskje tolkast i retning av at %INCLUDE er eit noko gammaldags val av makrobibliotek:

"The least sophisticated approach to setting up a macro library is obtained through the use of %INCLUDE files which store macro definitions. This was often the only viable type of library in earlier (pre V6) versions of SAS, and not a few SAS programmers have failed to make the transition to true macro libraries".

Den største ulempen med dette valet ser Carpenter å vera vedlikehold av filreferansar [17]:

"The greatest disadvantage of using the %INCLUDE in a large application is tracking and maintaining the filerefs that point to the individual files".

Eg har likevel valt det. Vedlikehold av katalognamn og filnamn er sentralisert og styrt ved hjelp av globale makrovariablar. Dessutan ser eg ikkje på Arkade_TestApp som nokon stor applikasjon. For ein så liten applikasjon har SAS ingen problem med det ekstra minneforbruket som trengs for å kompilera ein makro for kvar gong den skal brukast i ein ny sesjon (det held med kompilering ein gong per sesjon). Det er også ulemper med Compiled Stored Macros. For eksempel ved oppdatering av kode for ein makro, kan det vera fort gjort å gløyma å rekompilera koden før enn ein eksekverer den på nytt [17]:

"The compiled macros are all in one location, the SASMACR catalog, however the code itself could be anywhere. Usually developers that use compiled stored macro libraries will also have a central location to store the source code".

"When developing macros in the interactive environment, special care must be taken to make sure that the correct macro definitions are compiled and stored. If the developer is not careful it is possible to call a macro without using the latest update to the macro definition. This is not a good thing".

Kvar makro får si eiga programfil

Dette skaper større fleksibilitet og uavhengigheit mellom kompilering og eksekvering av makroane [17]:

"Generally a given file should not contain more than one macro definition, and the macro being called should not be called from within the included code".

"The first is important because if the file contains multiple macro definitions, then all the definitions must be loaded, and compiled, just to use one of the macros. Of course if all the macros will eventually be used this does not really matter".

8.3 Avgrensingar

Resten av dette kapittelet handlar om testinga av Arkade. Det kan difor vera på sin plass her å kort gjera greie for praktiske avgrensingar som er gjort.

Tida til disposisjon for arbeidet har avgrensa omfanget av testinga. For dei former for statisk og dynamisk testing som er nemnde i del-kapittel 8.1, er følgjande ikkje implementert:

- Testing av struktur og vedlikeholdbarheit i koden.
- Testing av eksekverbar kode frå ein makro ved hjelp av ein annan makro.
- Testing av returkode.
- Testing ved bruk av kode som avsluttar eksekveringa.
- Testing ved å la globale makrovariablar erstatta kommentar-teikn i programkoden.

Testing av kombinasjon av inndata i metadatafila avgrensar seg til 23 testtilfelle. Sjølv om teorien for parvis testing ikkje er implementert, kan det seiast for sikkert at 23 testtilfelle ikkje er nok for eit design etter testteknikken parvis testing. Viss eg hadde fått til å bruka nokon av dei programma som er referert i punkt 8.7.6, ville eg i det minste kunna sagt kor mange testtilfelle det ville vore bruk for.

Omtalen av dei ulike testteknikkane viser prinsippet for dei, men å implementera dei ville krevja grundig gjennomgang av heile applikasjonen på over 30.000 kodelinjer.

8.4 Testteknikkar for å testa kode på einheitsnivå

Det beste er om ein kan unngå at feil i programkoden oppstår. Tida for å oppnå det er best når koden blir laga første gong. Under utviklingsprosjekt skjer mykje av testinga på einheitsnivå av den som sjølv utviklar koden.

Felles for dei aktuelle testteknikkane er at dei blir brukt ved testing på lågnivå (einheitsnivå). Einheiten kan vera eit datasteg, ein makro eller ei programfil. Dei fleste av testteknikkane har som formål å redusera feil under kodinga eller oppdaga feil etter kodinga. Testteknikkane er:

- Testing av returkode.
- Testing ved bruk av kode som avsluttar eksekveringa.
- Testing av struktur og vedlikeholdbarheit i koden.
- Testing av gyldigheitsområde for markovariablar.
- Testing av eksekverbar kode frå ein makro ved hjelp av ein annan makro.
- Testing ved å la globale makrovariablar erstatta kommentar-teikn i kjeldekoden.
- Testing av korleis Arkade tek vare på evidens til bruk for å bli testa.
- Testing av bruken av makroparametrar i Arkade.

Desse teknikkane blir presentert i det følgjande. Dei medverkar til å gje svar på samtlege av dei spesielle problemstillingane:

- *Korleis bør Arkivverket sitt SAS-program Arkade testast?*
- *Kor lett testbart er dette programmet?*
- *Korleis kan det eventuelt gjerast meir testbart?*

8.4.1 Testing av returkode

I artikkelen ”Proactive Debugging” [102] skriv Ratcliffe:

”One of the ways that the task of bug fixing can be enhanced is to catch the problem as soon as it occurs. Unless every single return code is checked, a problem can continue until it causes a bigger problem at a later stage. This paper describes a quick and convenient method for testing and capturing every return code - and presenting meaningful information should a failure occur”.

Den aktuelle artikkelen inneheld koden på ein makro som er i språket SCL (SAS Component Language). Sjølv om Arkade brukar SCL ønskjer er heller tilsvarande kode i det tradisjonelle makrospråket, fordi eg beherskar det betre og mykje av arbeidet med testapplikasjonen har hatt som formål å frigjera seg frå SCL-rammeverket for å ha betre kontroll over testinga.

Eg har utvika eit knippe med makroar for ulike formål. Felles for desse er at dei i stor grad brukar returkoder. Vedlegg 6 viser programkoden for desse. Eksempler på føremål med utvikla makroar er:

- Testa om ein katalog finst.
- Testa om eit datasett eksisterer.
- Testa om eit datasett er tomt eller ikkje.
- Testa talet på observasjonar i eit datasett.
- Testa innhold i eit datasett ved skriva ut innholdet.
- Testa om ei fil eksisterer.
- Testa om ei fil har eit gyldig SAS-namn.

8.4.2 Testing ved bruk av kode som avsluttar eksekveringa

Ratcliff [103] skriv at

"There's nothing worse than a program that continues after a problem has occurred. The program will inevitably destroy evidence about the original cause of the problem. It will do this by over-writing variables and data sets, etc. If you want to get to the root of a problem, you need as much good evidence as possible".

Han viser eit kodeeksempel på at RUN CANCEL kan brukast til å unngå at vidare prosessering av koden etter at ein feil har skjedd slik at koden ikkje slettar spor bak seg som vil gjera debugging vanskelegare. Eit eksempel på dette er vist under.

```
%macro herbert;
    %let cancel = ;
    proc summary data = sashelp.class;
        class sex;
        var height wait;
        output out = summ sum =;
    run &cancel;

    %if &syserr ne 0 %then %let cancel = cancel;

    data result;
        set summ;
        if height gt weight then put 'h>w';
        else put 'w>h';
    run &cancel;

    %if &cancel ne %then %put Program did not complete successfully;
%mend herbert;
%herbert;
```

I prosedyren PROC SUMMARY blir variabelen WAIT referert utan at denne variabelen finst i datasettet SASHELP.CLASS. Dette fører til at den automatiske makrovariabelen &SYSERR

blir forskjelleg frå 0 og følgjeleg blir den brukardefinerte makrovariabelen &CANCEL tilordna verdien 'cancel'. Dette fører til at datasettet RESULT ikkje blir laga. Under er vist utdata frå SAS-loggen ved eksekvering av koden over:

```
579  %macro herbert;
580      %let cancel = ;
581      proc summary data = sashelp.class;
582          class sex;
583          var height wait;
584          output out = summ sum =;
585      run &cancel;
586
587      %if &syserr ne 0 %then %let cancel = cancel;
588
589      data result;
590          set summ;
591          if height gt weight then put 'h>w';
592          else put 'w>h';
593      run &cancel;
594
595      %if &cancel ne %then %put Program did not complete successfully;
596  %mend herbert;
597  %herbert;
ERROR: Variable WAIT not found.

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.SUMM may be incomplete.  When this step was
stopped there were 0
         observations and 0 variables.
WARNING: Data set WORK.SUMM was not replaced because this step was stopped.
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


WARNING: DATA step not executed at user's request.
NOTE: DATA statement used (Total process time):
```

Flavin og Carpenter [104] skriv at setningar som ENDSAS, STOP, RUN CANCEL, ABORT og ERRORABEND kan brukast i programlogikken til å avslutta ein modul eller heile sesjonen. Modul i denne samanheng kan vera eit datasteg, ein prosedyre eller ein makro. I forhold til Arkade ser eg det ikkje som ønskjeleg å bruka kommandoane ENDSAS, ABORT eller ERRORABEND med mindre det kan vera kode der ein kan gå inn i evige løkker. Men i slike tilfeller kan ein også bruka av-knappen på maskina. Ulempen ved å bli kasta ut av koden kan vera at ein mister muligheiten for å spora opp feilen. Ein kodeavslutning som kan vera aktuell for Arkade, er RETURN inni ein makro; den gjer at eksekveringa av makroen avsluttast, utan at sesjonen blir avslutta.

8.4.3 Testing av struktur og vedlikeholdbarheit i koden

Ratcliff meinar det er viktig å bruka ein kodestandard med tanke på at andre enn ein sjølv skal lesa koden seinare [103]:

"Don't underestimate the value of neatness in making your code more understandable for you and others.".

Såkalla ”match-koding” blir fråråda av Ratcliff på det sterkast [103]:

”We’ve all seen it! Functions embedded within functions embedded within functions, with buckets of parentheses thrown in for good measure; usage of esoteric bits of SAS functionality that were dropped from the documentation years ago but are still in the product; clever use of advanced mathematical functions to save using a larger number of basic mathematical functions. Don’t do it! Nobody will thank you when they have to maintain your program”.

Ratcliff påpeikar kor viktig det er å følgja kodestadardar, og at “vakker” kode er mykje enklare å vedlikeholda enn “stygg” kode. Med referanse til eigen erfaring med koding i SAS er det mi subjektive oppfatning at ”vakkert” er synonymt med ”symmetrisk”. Schlegelmilch [105] har ein del gode råd for korleis SAS kode bør strukturerast for å vera lett å vedlikeholda, sjå vedlegg 7. Dette vedlegget viser også eit eksempel på at Arkade har eit stort forbetringspotensiale når det gjeld strukturering av programkoden. Essensen er:

- Strukturerte innrykk i koden.
- Makroar bør framhevast med eigne innrykk.
- Avslutninga av ein makro bør ha referanse til starten av den.
- Strukturert bruk av STOR SKRIFT og lita skrift.
- Ikkje meir enn ein kommando på ei og same linje.
- Makrospråk og SAS-språk bør innordna seg i den same innrykkstrukturen.

8.4.4 Testing av gyldigheitsområde for markovariablar

Ratcliff [103] meinar makrovariablar helst bør vera definert med lokalt gyldigheitsområde. Sjølv forklarar han gyldigheitsområde slik:

”The “scope” of a variable is the range of sections within the program in which the variable is valid”.

Det er generelt ansett som god programmeringspraksis [106] å utvikla applikasjonar i makrospråket ved å bruka lokale makrovariablar der ein kan. Som påpeikt av Ratcliff [10] har dette nær samanheng med kor vidt ein legg forholda til rette for å lykkast med debugging av eit program:

”By restricting the life of information in this way, it becomes easier to trace the origin of information. Clearly it’s not good practice to have variables with the same name in both the global and local macro environment…”.

Eg har difor sett det som vesentleg å ha fokus på dette i evalueringa av Arkade. Om vi kallar default bruk av lokale makrovariablar for ein ”testteknikk” eller ikkje er underordna, det vesentlege er at dette påverkar muligheitene til å lokalisera feil i progamkoden. For Arkade sin del har eg forklart dette nærmare i vedlegg 8. Eg har utvikla to makroar for å testa gyldigheitsområdet for makrovariablar. Makroane samarbeidar med kvarandre (den eine

kallar på den andre), og kan samla sett sjåast på som ein testteknikk. Testteknikken er dokumentert i vedlegg 8.

8.4.5 Testing av eksekverbar kode frå ein makro ved hjelp av ein annan makro

Ved eksekvering av ein makro, vil kodegeneratoren produsera eksekverbar kode. Då vil eventuelle makrovariablar vera løyste ut til den verdien dei skal ha. Sjølv om SAS-loggen kan vera til hjelp for å forstå den koden som blir eksekvert, kan det i tillegg vera ei endå betre hjelp om ein fekk sjå denne reine, eksekverbare koden i editoren i SAS, fordi då ville ein få hjelp av editoren sine innebygde fargeeigenskapar til å oppdaga syntaksfeil i den eksekverbare koden. Det ville vera ein god testteknikk å bruka i debugging. Ratcliff [103] har påpeikt dette problemet:

"If you have a problem with a macro then things can get extra tricky because the macro debugging tools are so limited. I find it often helps to remove the macro code from the "equation" by copying the generated Base SAS code from the log to the editor (having set options mprint before running the code). However, if you have a lot of code being generated it can be tiresome to remove all those copied line numbers and other paraphernalia from the log. In this case you can use options mfile. Used in conjunction with the mprint option and an mprint filename statement, mfile gives you a "clean" copy of the code generated by your macro".

Det er ikkje uvanleg at ein makro i Arkade er på 2-3000 kodelinjer. For eksempel er makroen %GEN_XML på 2437 kodelinjer inkludert kommentarlinjer. Men sjølv for makroar på under 100 kodelinjer kan det vera ineffektivt å "rensa" den eksekverbare koden i SAS-loggen for informasjon om kodelinjenummer og anna tilleggsinformasjon som vert lagt på av SAS i loggfila. I vedlegg 9 har eg dokumentert ein makro eg har utvikla for å forenkla/effektivisera dette arbeidet. Makroen genererer ei fil som så kan limast direkte inn i editoren i SAS, og då vil fargekodinga lett avsløra kvar i den eksekverbare koden det er feil.

8.4.6 Testing ved å la globale makrovariablar erstatta kommentar-teikn

Ein hensikt med å laga kommentarar i kjeldekoden kan vera å dokumentera programmet. Eit anna formål kan vera å kommentera bort eksekverbar kode som ein ikkje ønskjer å sletta, fordi den kan brukast til testing av programmet. Det er det siste formålet som er relevant for den testteknikken Flavin og Carpenter [104] skriv om. Det kan vera tungvint og upraktisk å køyra fleire hundre (eller tusen) kodelinjer i strekk for debugging-formål, når ein vanlegvis køyrer koden for produksjons-formål. Enten må ein då gå inn i koden og fjerna kommentarane før ein køyrer i debugging-modus og setja dei på att etter ein er ferdig med debugginga, eller så må ein køyra små kodebitar i gangen manuelt og passa på at kompilatoren unngå å støyta på dei token som gjeld for kommentering. Flavin og Carpenter [104] viser korleis ein kan bruka makrovariablar i stadan for å hardkoda tokena for kommentarar. Det er tre former for kommentarar i SAS, og alle tre kan brukast både i SAS-språket og makrospråket. Dei tre måtane er vist under:

```
Data test;
```

```
  Set sashelp.class;
  *if age > 13 then group = 'adult';
  /*if age <= 13 then group = 'child';*/
  %*if age < 3 then group = 'baby';
run;
```

Forfattarane skriv ingenting som tyder på at desse formene for kommentarar ikkje er like gode, men det er dei ikkje [106]. Den absolutt sikrast forma for kommentarar er

/* */. Den som er minst sikker er %*. I tråd med dette vil eg endra dei eksempla som er gjeve i artikkelen, slik det framgår av eksempelkoden under:

```
%let checkit_slash = /;
%let checkit_star = *;
%let checkit_start = &checkit_slash.&checkit_star;
%let checkit_end = &checkit_star.&checkit_slash;

%let checkit_start = ;
%let checkit_end = ;

data test;
      set sashelp.class;
      &checkit_start;
      if age > 13 then group = 'adult';
      if age <= 13 then group = 'child';
      if age < 3 then group = 'baby';
      &checkit_end;
run;
```

No vil if-setningane eksekverast eller ikkje avhengig av valt verdi for makrovariablane &CHECKIT_START og &CHECKIT_END. Viss ein er strukturert når ein skriv koden, og tenkjer framtidig testing/debugging, vil det vera enormt arbeidsbesparande å kunna endra mellom debugging-modus og produksjonsmodus for kommentarar berre ein stad i programmet.

I vedlegg 10 viser eg eit ekesempel frå koden i Arkade på korleis denne testteknikken kan brukast.

8.4.7 Testing av korleis Arkade tek vare på evidens til bruk for å bli testa

Ratcliffe [103] påpeikar at det å ta vare på evidens, som han kallar det, er viktig. Med det meinar han å ikkje laga datasett som overskriv kvarandre:

"If you have to investigate a problem, you will want to follow the chain of events backwards to find the original cause. You won't be able to do this if some of the information in that chain is missing".

"I never overwrite intermediate (temporary) data sets. Those data sets may not be important in terms of your final results, but they are important if you are debugging. If you call your intermediate data sets something like "temp" then it's preferable to create temp1 from your input data set, and create temp2 from temp1. If you create temp and then overwrite it then you have lost the "paper trail".

This approach can result in a large number of TEMPnn data sets, so you should delete these data sets when they are no longer required, e.g. delete temp1 after temp2 has been created from it, otherwise your work library may run out of space. But if you delete these data sets you've lost your paper trail! So, I make the deletion conditional upon a macro variable that indicates whether I am in debug/development mode or in production mode".

Å la sletting av midlertidige datasett vera vilkårsstyrt av ein makrovariabel med ulik verdi avhengig av om ein eksekverer koden i debugging modus eller produksjonsmodus, er svært god. Ideen er heilt i tråd med mi valte tilnærming om å setja systemopsjonar for desse to modi.

Vedlegg 11 evaluerer Arkade i forhold til denne testteknikken. Konklusjonen min er at Arkade ikkje i tilstrekkeleg grad tek vare på evidens for å bli testa godt nok.

8.4.8 Testing av bruken av makroparametrar i Arkade

Testteknikken som er brukt for denne testinga, er kodegjennomgang.

Det er to hovudforhold som er evaluert i dette vedlegget:

- Type parametrar som er brukt.
- Dokumentasjon og validering av parametrar.

Testinga er nærmare beskriven i vedlegg 12. Arkade har 136 makroar. 91 av desse har ingen parametrar, 45 har parametrar. Alle dei 45 makroane som har parametrar, har reine posisjonsparametrar.

Resultat av testinga er at i praksis har eg ingenting å utsetja på bruken av makroparametrar i Arkade. For makroen %GET_HEX konkluderer eg med at dokumentasjonen i kjeldekoden er for dårleg.

8.5 Testteknikkar for debugging ved hjelp av innebygd funksjonalitet i SAS

Aktuell innebygd funksjonalitet i SAS er gruppert slik i vedlegg 13:

- Systemopsjonar relevante for analyse og debugging av SAS-program.
- Systemopsjonar relevante for analyse og debugging av makrospråk.
- Datasteg opsjonar relevante for analyse og debugging av datasteg.
- Automatiske makrovariablar relevante for debugging av makrospråk.
- Automatiske variablar relevante for debugging av datasteg.
- SAS-setningar relevante for debugging av datasteg.

- Prosedyrer relevante for debugging av SAS-program.
- Funksjonar og call-rutiner i SAS.
- Metadatatabellar (datasett) i SAS relevante for debugging av SAS-program.
- Editoren i SAS.

Vedlegg 13 gjev nærmare greie for nærmare detaljar for denne funksjonaliteten.

På grunnlag av resultata som er dokumenterte i vedlegg 13, har eg funne at Arkade har følgjande forbetringspunkt:

- Systemopsjonen SOURCE2 bør takast i bruk.
- Systemopsjonen DATASTMTCHK bør setjast til ALLKEYWORDS.
- Systemopsjonen MLOGIC bør enkelt kunna veljast framfor NOMLOGIC.
- Systemopsjonen MPRINT bør flyttast frå fila MIXC_DSE.sas til AUTOEXEC.sas, men ei forklaring i kjeldekoden på bruken av den.
- Systemopsjonen MLOGICNEST bør takast i bruk ved testing av Arkade.
- Systemopsjonen MPRINTNEST bør takast i bruk ved testing av Arkade.
- Systemopsjonen SYMBOLGEN bør enkelt kunna veljast ved testing av Arkade.

Dette tyder ikkje at det ikkje kan vera anna innebygd debuggingsfunksjonalitet som kan utnyttast betre. Testinga av Arkade har ikkje vore intensiv på dette området.

Val av mange av systemopsjonane innebygd i makroen %SASLOG_VELG_OPSJONER som er ein del av testteknikken for kvalitetssikring og effektivisering av gjennomgangen av SAS-loggen. Denne testteknikken blir gjennomgått i det følgjande del-kapittel.

8.6 Testteknikk for analyse av SAS-loggen

I dette del-kapittelet fokuserer eg på følgjande:

- SAS-loggen si rolle for debuggingsprosessen.
- Vurdering av testteknikkar for kvalitetssikring og effektivisering av gjennomgangen av SAS-loggen som andre har laga.
- Mi eiga løysing for testteknikk for kvalitetssikring og effektivisering av gjennomgangen av SAS-loggen.

8.6.1 SAS-loggen si rolle for debuggingsprosessen

Viss ein var nøydd til å rangera ulike hjelpemiddel for å debugga eit SAS-program effektivt, tyder mykje på at det å få kontroll over SAS-loggen ville vera øverst. Delwiche og Slaughter legg stor vekt på SAS-loggen [107]:

"The first and most important rule in debugging SAS programs is to always, always check the SAS log. After running a SAS program many people turn immediately to the output. This is understandable, but not advisable. It is entirely possible – and sooner or later it happens to all of us – to get output that looks fine but is totally bogus. Often, checking the SAS log is the only way to know whether a program has run properly."

Som eitkvart hjelpemiddel må også SAS-loggen brukast på rett måte.

Når eit program har køyrt, viser loggvindauget i SAS slutten av loggfila. Det er viktig å starta gjennomgangen av loggen frå starten [107]:

"The second most important rule in debugging is to always start at the beginning of the log. This may seem obvious, but if you are running SAS in the windowing environment, then the SAS log fills up the Log window and you are left looking at the end of the log, not the beginning. There is a temptation to try to fix the first error you see, which is not necessarily the first error in the log. Often one error early in the program can generate many messages, and only the first message will give you a clue as to what the problem is."

I tillegg bør loggen frå forrige programkøyring blankast ut når ein er ferdig med å analysera den, for at ein lett skal vita kvar i loggen den neste køyringa startar.

Å undersøkja SAS-loggen er eit relativt uttrykk. Alt frå å kasta eit raskt blikk på den til å undersøkja svært grundig. Det er fort gjort å ta for lettvint på SAS-loggen. I Mitchell sin artikkel [108] under avsnittet "Did you really look at that SAS log?", har han synspunkt på dette:

"So the program ran and output was generated – oh yeah, man! We're done now? Wrong. Especially to the beginning programmer and perhaps even the overconfident veteran programmer, the very fact that output was produced can give one a false sense of security, and the SAS Log has the potential to be taken for granted either by the overconfident senior programmer or the unsuspecting junior programmer."

Når det gjeld raude ERROR-meldingar i SAS-loggen, skriv Mitchell:

"Reviewing the SAS Log for ERROR messages should be a standard that all SAS programmers should follow."

Når det gjeld grøne WARNING-meldingar i SAS-loggen, skriv Mitchell:

"While WARNING messages are not considered as severe as ERROR messages in terms of stopping certain parts of a program, these messages can certainly shed light on potential significant issues. One should take all WARNING messages seriously. An example

of this is the "repeats" of BY variables warning that lets one know that the program is trying to merge by too many common variable attributes. Unfortunately, the program will continue and decide as best as it can how certain items should be merged, but the vast majority of the time, this will result in improper matching of data."

Sist, men så absolutt ikkje minst, den størst faren ligg kanskje i det som har blå farge:

"Believe it or not, there are other reasons why one would look at the NOTE messages in the SAS Log besides knowing that SAS took X minutes and seconds to run this DATA step and Y minutes and seconds to run this particular procedure. Additionally, just because all notes appear to be okay, again, this absolutely does not mean that everything IS okay. One might use a certain level of judgment when reviewing these notes with careful attention regardless of the length of the program. Beware of DATA steps that result in "0 observations" as these can cause the program to appear to be running smoothly with even a reasonable output being generated, but without careful review one would be totally oblivious to whether the results are actually accurate or not."

Howard og Gayari [109] sin artikkel understøttar at NOTE-meldingar i SAS-loggen byr på mange høve til nyttig informasjon:

"However, much useful information can be gleaned from reading the NOTEs in the SAS Log -these are warning or informational messages that do not stop processing although they may point to conditions that affect the accuracy of program results. For example, numeric over- or under-flow, taking logarithms of zero or negative values, uninitialized variables, invalid data, insufficient format widths, hanging DOs or ENDs, missing values, and character- to-numeric or numeric-to-character conversions are all conditions that generate NOTEs in the log.race), thus saving PROC SORT time. "

8.6.2 Vurdering av testteknikkar laga av andre

Eg har undersøkt kva som er gjort tidlegare av løysingar for parsing av SAS-loggen, og vurdert om eg kan bruka noko av dette til testing av Arkade.

Tabellen under viser kva for forslag eller idear til løysingar som er vurdert, og korleis eg har vurdert dei:

Løysing av	**Ref.**	**Kode presentert?**	**Kode som fungerer?**	**Vurdering**
Stojanovic	[110]	Ja	Ja	Det var god praktisk hjelp å få av artikkelen til Stojanovic. Rapporten frå parsing av SAS-loggen har ein tabell med kolonnene 'Type', 'Line # in SAS LOG' og 'Original SAS log message', og desse fall i smak. Programmet søker i SAS-loggen på orda ERROR, WARNING og NOTE. Eg kunne i tillegg tenkt meg at det vart søkt på INFO. Og eg ville gjerne hatt presisert kva for NOTE det skal søkast på. Dei fleste av NOTE meldingane er ikkje

				verdt å ha med. Det bør vera ei eksplisitt liste av kva for NOTE meldingar ein vil ha med, elles vil ekstraktet av informasjon bli for stort. Eg kunne også tenkt meg å hatt med meldingar av typen _ERROR_, som er datasteget sin måte å rapportera til SAS-loggen om datafeil på. Fordi datafeil kan vera like interessante å fanga opp som programfeil.
Smoak	[111]	Nei	-	Forklarar på eit overordna plan. Lite praktisk hjelp.
Fehd	[112]	Ja	Nei	Nyttige tips om korleis innparametrar kan brukast til å angje type melding frå SAS-loggen det gjeld. Rapport i html-format, skulle gjerne hatt den i word.
Terjeson	[113]	Ja	Ja	Rapporten manglar referanse til kvar i SAS-loggen feilmeldinga er (linjenummer). Meldingar av typane NOTE og INFO blir ikkje tekne med.
Yellanki	[114]	Nei	-	Nyttig oversikt over typar av feilmeldingar som rapporten bør fanga opp frå SAS-loggen.
Buck og Stewart	[115]	Nei	-	Nyttige tips til feilmeldingar som rapporten bør fanga opp frå SAS-loggen.
Truong	[116]	Ja	Nei	Lite nyttig. Ingen kode i sjølve artikkelen.
Li og Troxell	[117]	Ja	Nei	Artikkelen gjev god motivasjon for å utvikla ein makro for å finna viktige feilmeldingar i SAS-loggen. Koden er ikkje fullstendig dokumentert i artikkelen. Uråd å få koden frå forfattarane direkte.
Humphrey	[118]	Ja	Nei	Artikkelen inneheld mange konkrete eksempler på kva for feilmeldingar i SAS-loggen som blir fanga opp av koden. Det er også utskriftseksempler på rapport frå parsing av loggen. Det blir to rapportar for kvar parsing, ein for akkumulert informasjon (talet på ulike typar feilmeldingar) og ein rapport for nedbrytning til dei einskilde feilmeldingane. Eg ønskjer begge typar informasjon, men eg ønskjer dei i ein og same rapport.
Foley	[119]	Ja	Ja	Det leggest opp til at alle NOTE-meldingar skal rapporterast. Det blir altfor mange. Godt forslag til implementering av ein makro for sjekk av SAS-loggen.
Sherman	[120]	Ja	Nei	5 makroar. Tilsaman ca. 160 kodelinjer. Forsøket på å få koden til å fungera, mislykkast.

Tabell 8: Vurdering av testteknikkar for kvalitetssikring og effektivisering av gjennomgangen av SAS-loggen

For nærmare omtale av dei ulike løysingane visest til vedlegg 14.

8.6.3 Mi eiga løysing for analyse av SAS-loggen

Testteknikken inneheld 2 formater, 16 makrovariablar og 10 makroar, som alle er utvikla i SAS. Når formata, makrovariablane og makroane samarbeidar med kvarandre, kan dei sjåast på som ein testteknikk.

Både av plasshensyn og fordi denne testteknikken er sterkt teknologisk fokusert er dokumentasjonen plassert i vedlegg 15.

Koden eg har utvikla er inspirert av dei ovanfor refererte kjeldene. Koden er implementert i testprogrammet "Arkade_TestApp". Vedlegget beskriv dei ulike delane av testteknikken og gjev eit eksempel på korleis ein rapport etter parsing av SAS-loggen ved bruk av testteknikken ser ut. Testteknikken har også innebygd funksjonalitet som gjer at dei loggmeldingar som kjem til det defaulte loggviewet i SAS Base lett kan sporast tilbake til den eksekverbare programfila loggmeldingane stammar frå.

8.7 Testteknikken parvis testing for design av utval av testtilfelle

Tidlegare i dette kapittel vart testtilfelle valt som rammeverk for den dynamiske testinga. Det tyder at testtilfelle står sentralt i testplanlegginga. I kapittel 5 er "Estimering og testplanlegging" nemnd som eit av "andre faktorar enn testmetoden i seg sjølv" som er viktige for om feil blir oppdaga.

Det er naturleg å starta med evaluering av tidlegare testplanar for Arkade, fordi det er grunn til å tru at andre har tenkt kloke tankar før meg, og også gjort erfaringar som eg kan dra nytte av.

Inspeksjon av dokument ligg til grunn for evaluering av tidlegare testplaner for Arkade

8.7.1 Evaluering av tidlegare testplanar for Arkade

Det foreligg interne dokument i Riksarkivet som gjeld eller er testplanar for Arkade. Dei tre første er frå 1999 [121] [122] [123]. Utviklaren av programmet har laga eit forslag til testplan [124] datert 2001 og prosjektleiar har laga utkast til testplan [125] i 2002. Dei internt produserte utkasta er merka utkast 4, 5 og 6 i sjølve dokumenta.

Felles for alle testplanane er at dei legg opp til testing av ulike kombinasjonar av inndata. Testteknikken er svartboks testing (vedlegg 1, eller sjå [46]). Det eine av dokumenta [123] er slik eg forstår det eit dokument basert på to føregåande dokument [122] og [121], og legg opp til sju "funksjonar" som skal testast:

- Innlesing og tolkning av XML-fil
- Syntaks
- Semantikk/konsistens

        - Innlesing og intern representasjon av data frå tekstfiler
    - Samsvar mellom XML-fil og datafil
    - Sjølve innlesingen frå tekstfiler til SAS
        - Sjekking av oppgjeve strukturinformasjon
        - Analyse av datafilenes struktur
        - Konvertering av data til nye tekstfiler
        - Produksjon av ny XML-fil
        - Oppretting av SAS-datasett

Testplanen frå 2002 [125] legg opp til fire ”funksjonar” som skal testast:

1. Innlesing og tolkning av XML-fil og struktur i datafil
2. Oppretting av SAS-datasett.
3. Konvertering av data til nye filer med ny XML-fil
4. Analyse av datafila sin struktur og felt

Eg vil gå nærmare inn på desse to testplanane med tanke på kva eg vil trekka med meg til design av testplan og testtilfelle for Arkade no.

Testplanen frå 2002 [125] brukar uttrykka ”strukturkritiske”, ”tolkningsbetinga” og ”uavhengige” felt. Av samanhengen desse uttrykka blir brukt i, forstår eg det slik at ”strukturkritiske” felt er felt med inndata som må testast godt, og som må bli prosessert feilfritt utan forbehold. For ”tolkningsbetinga” felt legg testplanen lista lågare for kor grundig inndata treng testast. Testplanen forklarar ikkje kva ”uavhengige” felt tyder. Bortsett frå felt ’Datasett namn’, ’Logisk filnamn’ og ’Fysisk filnamn’ kan alle dei ”uavhengige” felta ha blank verdi, men om det er det som har kvalifisert til betegnelsen ”uavhengig” er vanskeleg å sei. Det er også mange av dei ”tolkningsbetinga” felta som har blank som lovleg dataverdi. Dei såkalla ”strukturkritiske” felta er Teiknsett, Format, Postskilje og Versjon, alle på datasett-nivå i metadata-fila. Desse felta kan ikkje ha blanke dataverdiar.

Kor vidt eit felt karakteriserast som ”strukturkritisk”, ”tolkningsbetinga” eller ”uavhengig”, meinar eg er irrelevant for design av testtilfelle, og følgjeleg også for testplanen. Om eit felt er obligatorisk eller ikkje, gjev ingen informasjon om sannsynlegheiten for at Arkade vil få feil ved prosessering av filer.

Begge testplanane har ’Oppretting av SAS-datasett’ som funksjonalitet ein vil testa på om skjer. Eg ser ikkje nokon god grunn til å velja generering av datasett bort ved opsjonen i grensesnittet til Arkade. Å få datasett generert, er berre fordelaktig fordi inspeksjon av dei kan gje eit første visuelt inntrykk av om innlesinga av filene til SAS har gått greit. Eg vil satsa på inspeksjon av SAS-loggen for å avdekka at eit datasett ikkje blir oppretta som følgje av ein

feil i programmet. Det å ta dette punktet bort frå design av testtilfelle, tyder ikkje at eg ikkje vil bruka inspeksjon av datasett til å kontrollera at innlesinga har skjedd rett. Det tyder berre at eg meinar det ikkje er er ein interessant eigenskap ved Arkade om den greier la vera å laga eit datasett når opsjonen er å ikkje generera eit slikt. Det er overflødig funksjonalitet, slik eg ser det. Testplanen frå 1999 [123] skriv at :

"Selve opprettelsen av SAS-datasett kan testes ved å studere de SAS-lagrede dataene visuelt ved hjelp av SAS' egne datafremvisningsmoduler…" "Denne uttestingen bør gjøres grundig tidlig i den totale testprosessen, slik at oppretting av SAS-datasett kan benyttes som testverktøy i fortsettelsen. Også ved opprettelse av SAS-datasett vil det være aktuelt å studere loggen fra SAS for å se om det dukker opp "Errors" eller "Warnings". "

Etter mitt syn er det viktigaste å få testa ut *korleis Arkade reagerer på samsvar eller mangel på samsvar mellom metadata-fil og datafil(er)*. Viss det ikkje er samsvar mellom metadata-fila og datafil(ene), kan ikkje SAS vita for sikkert kven som er feil. Det kan vera den eine, eller den andre, eller begge. *Det som er viktig, er at SAS reagerer på divergensen. Men forventningane om korleis feil skal avdekkast må avstemmast etter kva for feil det gjeld*. Viss det er metadata-fila som har oppgjeve feil startposisjon for eit felt i ei fil med fast postlengde, og postlengden likevel er fast, kan vi ikkje forventa verken "Error", "Warning" eller "Note" frå SAS-loggen. Denne type feil vil best kunna oppdagast ved inspeksjon av feltverdiar i datasett. Det føreset at det ikkje er andre feil som har stansa innlesinga i datasettet. Då må slike feil i så fall rettast først. SAS vil heller ikkje reagera med melding i SAS-loggen dersom ein post har for mange felt i datafila, men likevel held seg til den faste postlengden.

Ved konvertering genererer Arkade både ny datafil og ny metadata-fil som utdata. Testplanen frå 1999 [123] gjev greie praktiske råd for korleis ein kan testa om utdata vart som forventa: Ein editor, ein xml-parser, program som "fileCompare" eller "CompareDiff", eller manuell inspeksjon av data i datasett.

Testplanen frå 1999 [123] inneheld 5 kategoriar av feil og manglar: Kritiske feil, alvorlege feil, uvesentlege feil, midlertidige feil og permanente feil. Som forbetring av kategoriseringa vil eg foreslå å redusera til dei tre førstnemnde. Grunnen er at dei to siste kategoriane er naturlege underkategoriar av den tredje.

Dei to testplanane er noko ulike når det gjeld praktisk testgjennomføring. Testplanen frå 1999 [123] verkar meir målretta, fordi den er konkret på mange forhold som bør testast, og den har ei innstilling som i sterkare grad reflekterer holdningen "destruktiv testing". Testplanen frå 2002 [125] viser i punkt 5 ein tabell over aktuelle kombinasjonar av felt som bør testast. Slik eg tolkar denne tabellen, reflekterer den eit kartesisk produkt av alle kombinasjonsmuligheiter. Ein skal ikkje ha så mange parametrar før dette blir urealistisk å gjennomføra i praksis. Mitt forslag til endring av testdesign er parvis testing [72].

Systemutviklar har laga eit "Forslag til testsystem" frå 2001 [124]. Mitt umiddelbare inntrykk er at dette må det vera vanskeleg for ein kunde å ha noko meining om når det gjeld behov og omfang. Det tyder på at ein bør kjenna strukturen i programmet godt for å ha utbytte av denne testplanen. Denne testplanen har heller ingen prioritering av kva ein bør testa først.

Oppsummering av kva eg ønskjer å behalda:

- Grunntanken med testing av inndata frå metadatafila.

- Inspeksjon av datasett som ein viktig testteknikk.

Oppsummering av kva eg ønskjer å bidra med:

- Kutta ut testing av om datasett ikkje blir oppretta viss opsjonen i Arade sitt grensesnitt tilseier det.
- Sjå bort frå irrelevant kategorisering av inndata.
- **Fokus på feila sin natur framfor kategorisering av feil.**
- **Parvis testing som testteknikk for design av utval av testtilfelle.**
- **Prinsippskisse for å illustrera effekten av parvis testing.**
- **Grunnlagsskisse for design av testtilfelle ved testing av Arkade.**
- **Implementering av testteknikk for parvis testing.**

I resten av dette del-kapittelet gjer gjer eg nærmare greie for dei fem siste kulepunkta.

8.7.2 Fokus på feila sin natur framfor kategorisering av feil

Inndelinga av feil etter alvorlegheitsgrad er vanleg ved testing. Artikkel 5 [33] gjev ei oversikt over ulike måtar det er vanleg å klassifisera feil på:

"In this article, we first classify defects on the basis of their origin: requirements, design, or code. Second, many empirical studies categorize defects along two dimensions, as Victor Basili and Richard Selby proposed.The first dimension classifies defects as either an omission (something is missing) or a commission(something is incorrect), while the second dimension defines defect classes according to their technical content. Other classifications focus on the defect's severity in terms of its impact for the user: unimportant, important, or crucial".

Det er naturleg at Riksarkivet som kunde er interessert i å vita om feil som blir funne, er uvesentlege, viktige eller kritiske. Ei slik kategorisering kan tjena som ei prioriteringsliste for retting av feil. Men eg meinar det bør fokuserast meir på feila sin natur enn på kategorisering av feila, fordi det skaper større forståing for kvifor feila oppstår og kva for konsekvensar dei kan få.

Det eg har å forholda meg til, er ei feilliste frå det tidlegare testarbeidet med Arkade, utarbeidd og vedlikeholdt av den eksterne utviklaren av systemet [126]. Dokumentet er ein tabell i word med 11 kolonner og omtrent 70 rader. Kolonnene er :

- Feil nr
- Test-nr
- Dato
- Sign

- Type
- Status
- Kommentar
- Oppfølging
- Dato retta
- Retta i Arkade-versjon

Eg har nokre ankepunkt mot dette testopplegget:

- Feillista dokumenterer både feil og endringar. Difor blir dokumentet ei underleg blanding av ein "kravspesifikasjon", oversikt over feil, oversikt over endringar og eit feilhåndteringssystem.
- Feillista dokumenterer ikkje i kva for versjon av programmet feilen vart oppdaga. Det er kun dokumentert i kva versjon av Arakade feilen er retta.
- Feltet "Test nr" er ikkje fylt ut for 25 av radene i tabellen (feillista). Det tyder at nesten ein tredjedel av feila eller endringsforslaga ikkje er forankra i nokon test. Repetisjon av denne testen på eit seinare stadium for å finna ut om den same feilen fortsatt er der, blir dermed uråd.
- Feillista er inkonsistent for 21 rader. For 16 rader består inkonsistensen av at feltet "Dato retta" er utfylt til tross for at feltet "Status" er satt til "Henlagt". For 5 rader gjeld tilsvarande der feltet "Status" er satt til "Utsatt".

Feltet "Type" kan ha verdiane:

- F1 – Alvorleg feil.
- F2 - Mindre alvorleg feil.
- F3 - Kosmetisk feil.
- E - Endring.
- U - Usikker.

Dokumentet er ikkje sjølvforklarande for kva "Endring" tyder, men eg går ut ifrå at det gjeld eit nytt krav oppdaga under testing av systemet. Og eg er usikker på kva verdien "Usikker" tyder. Kanskje tyder det at den som har registrert verdien er usikker på val av kategori. Verdien "Usikker" er brukt for 3 av radene i tabellen i kolonna "Type". For to av desse radene er verdien i kolonna "Status" satt til "Henlagt", i den tredje raden er den "Registrert".

Statusfeltet kan ha verdiane:

- Registrert

- I arbeide
- Testast
- Må utredast
- Overførest
- Overført <dato>
- Gjenoppteke
- Utsatt
- Henlagt
- Ikkje noko gjerest

Verdien "Overført <dato> tyder at retting av feilen eller innfriing av kravet blir overført til neste versjon av Arkade.

Verdiane i feltet "Type" er heller ikkje brukt konsekvent. Tabellen under viser ei frekvensoversikt over verdiar i feltet "Type", samt ei gruppering av kva for status desse feiltypane har:

Type	**Tal postar i feillista**	**Registrert**	**Henlagt**	**Utsatt**	**Overført <dato>**
F1 – Alvorleg feil	29		10	2	17
F2 - Mindre alvorleg feil	10		1	1	8
F3 - Kosmetisk feil	2		1		1
E - Endring	11	1	4	2	4
U - Usikker	3		3		
F1/E	3				3
F2/E	1				1
E/F1	1		1		
F1/E ????	1		1		
F1??	2				1
F1/2	1				1
?	2			1	1
F1/F2	1				1

Tabell 9: Kategorisering av feil ved testing av Arkade utført då Arkade vart utvikla

Eg tolkar dei 8 siste postane i tabellen over som uttrykk for uvisse om kor alvorleg feilen skal klassifiserast. Eller uvisse om det er ei endring (kravspesifikasjon) eller ein feil ein står overfor.

Ei forklaring på at ein tjedjedel av alvorlege feil blir henlagt (10 av 29) kan tyda på at den som har testa ikkje har forstått om feilen ein har registrert er alvorleg. Det var den som identifiserte feilen som klassifiserte den, og utviklaren som registrerte status for feilen. For 2 av desse 10 som vart henlagt, vart status endra til "Henlagt" fordi feilen vart retta i neste versjon (den til eikvar tid aktuelle utviklingsversjon) av Arkade. Dette framgjekk av teksten i feltet "Oppfølging".

Hovudinntrykket mitt av feillista for Arkade er at dokumentet er vanskeleg å trengja inn i for den som ikkje har detaljert kjennskap til programmet sin indre struktur. Det kan godt henda at dei som testa hadde det, men det blir eit presentasjonsproblem når eit felt i ein word-tabell skal inneholda såpass mykje tekst som det fort kan bli snakk om viss ein forsøker å gje ei uttømmande forklaring på ein feilsituasjon. For eksempel er følgjande forklaring på eit fenomen pressa inn i kollona "Forklaring" i tabellen i feillista:

"Mekanisme for håndtering av temporære filer ved pre-prosessering virket ikke dersom det var mer enn en fil, og dessuten lengdeberegning av feltverdier.

MG_I_XML:

Etter at enkeltkarakterer for postskille er hentet frem i makroen CH_PARAM, beregnes nye variable:

FI_TMP_N : <Navn for temporær fil>

FI_TMP_P : <Bane for temporær fil>

FI_LF_FR : <Postskille i innlest fil>

FI_LF_TO : <Postskille etter prepros.>

i metadatasettet for filer - _DIC_FIL.

MG_SET_P:

Etter at enkeltkarakterer for postskille er hentet frem i makroen CH_PARAM, beregnes nye variable i metadatasettet for filer - _DIC_FIL.

BASE_FIL:

Bane for temporær fil samt karakterer for postskille i inndata-fil er nå lagt inn i metadatasettet for filer - _DIC_FIL. De benyttes for generering av setningstypene:

FILENAME_IN : filnavn for innleste data

PRE_CHANGE_CRLF : utfører preprosessering

DELETE_TEMP_FILE: fjerner temporær fil

Etter fjerning av temporære filer kommer det nå en melding i loggen som forteller at filen er fjernet".

Det blir eit praktisk og pedagogisk problem i eit utviklingsprosjekt når ein med så enkle hjelpemidler som ein word-tabell forsøker å løysa kommunikasjonsutfordringar av den karakter som her er snakk om.

Eg ser ein god grunn til å stilla spørsmål ved om det ikkje burde vore eit versjonshåndteringssystem til hjelp for både utvikling og testing av Arkade. Slikt er vanleg i dag ved utvikling av applikasjonar. Behovet kan sjølvsagt variera avhengig av storleiken på programmet. Denne feillista, som er på 15 ståande A4-ark, tjente som kommunikasjonsverktøy ved møter mellom utviklaren og tilsette i Arkivverket som bidrog til testinga av systemet.

Den eine av dei to feila som blir karakterisert som kosmetisk, forstår eg ikkje kvifor er kategorisert som den er. I feltet "Kommentar" står følgjande:

"Benytter en eksisterende data og ADDMML.fil for å lage en testfil for test 41. Dette innebærer å kjøre datafilen gjennom SAS uten å gjøre annet enn å endre formatet. Filen har egendefinert tegnsett (Sintran). Under innlesing av XML-fil får jeg følgende melding (advarsel):Ikke-ekstern(e) fil(er) mangler tilhørende datafil.Meldingen er for meg kryptisk, hva betyr den???"

I feltet "Oppfølging" står følgjande:

"Mer utfyllende meldingsliste kommer i loggen".

Denne feilen er henlagt, utan at det er dokumentert kva som var problemet.

Av dei 70 radene i tabellen i feillista, er det berre 7 som refererer til feilmeldingar i SAS-loggen. Det skal ikkje så veldig mykje dataprosessering til i SAS før enn det blir generert nokre tusen logglinjer. For personar som ikkje kan SAS godt vil mange feilmeldingar verka uforståelege. På denne bakgrunn synest eg at det er eit lågt tal når SAS-loggen blir referert kun i 10% av radene i feillista. Dette kan tyda at dei som testa Arkade meinte dei hadde gode føresetnader for å tolka meldingar i SAS-loggen. Eller det kan tyda at dei ikkje hadde faglege føresetnadar for å bruka SAS-loggen for å finna feil.

Både i artikkel 1 og 2 finn eg støtte for synet om at det bør fokuserast meir på feila sin natur enn på kategorisering av feila. I kapittel 5 skriv eg under avsnittsoverskrifta "Den enkelte feil sin natur":

"Av artikkel 1 framgår at det er den enkelte feil heller enn den type/kategori feilen måtte vera klassifisert under, som avgjer om feilen er lett å oppdaga eller ikkje. Dette funn vert støtta av artikkel 2 og artikkel 5".

I artikkel 2 uttrykkest dette slik (mi utheving) [30]:

"The results reported here appear to show that the different techniques have different strengths and weaknesses in terms of the faults that they help to uncover. ***Their absolute effectiveness as well as their relative effectiveness depends on the nature of the programs and more specifically on the nature of the faults in those programs****. This is supported by the significant body of related work which shows no consistent pattern in terms of absolute or relative effectiveness. Rather, as the programs and faults vary so do the results".*

Viss vi forstår betre feil som oppstår, kva som er dei vanlegaste forekommande feila i SAS sitt programmeringsspråk, og korleis SAS forsøker å fortelja oss kva som er gale, trur eg feilsøking og feilretting vil bli meir effektiv. Eg trur det vil gje betre avkastning for innsatsen enn å gruppera feil slik som vart gjort i feillista for Arkade. Det er ikkje gale å gruppera feil etter alvorlegheitsgrad, men det er ikkje nok. I ein debuggingsprosess er dei enkelte feila i seg sjølv meir interessante enn det å plassera dei i ei eller anna gruppe for eit eller anna godt formål.

Vedlegg 4 inneheld 2 eksempler på gruppering av feil i SAS:

- Eksempel 1 presenterer først ei gruppering presentert i ein artikkel av Buck og Stewart [115]. Denne presentasjonen tek utgangspunkt i korleis feila presenterer seg ved meldingar i SAS-loggen. Presentasjonen har fokus på feila sin natur heller enn på å kategorisera dei.

- Eksempel 2 presenterer ei gruppering slik den framgår av ein artikkel av Knapp [127] og Burlew [128]. Denne gruppering er vist i tabellen under.

Type feil	**Beskriving**	**Kan feilen finnast automatisk av SAS?**
Syntaksfeil	Setningane er ikkje i samsvar med reglane for språket.	Ja
Semantiske feil	Strukturen til programsetningane er feil.	Ja
Køyretids-feil	Feil som blir oppdaga når kompilerte program blir eksekvert.	Ja
Data-feil	Feil i datafilene som programmet skal prosessera.	Ja
Logiske feil	Programsetningane fører til eit anna resultat enn forventa.	Nei

Tabell 10: Kategorisering av feil i SAS-program ifølgje Burlew

Det mest interessant i tabellen, er etter mi meining at SAS automatisk kan finna alle andre feil enn logiske. Indirekte stadfestar det at SAS-loggen er viktig.

Delwiche og Slaughter [107] brukar ei klassifisering som nesten er heilt lik med den i tabellen over:

"One way of classifying computer bugs is to divide them into three types of errors: syntax, data, and logic. Syntax errors result from failing to follow SAS's rules about the way

keywords are put together to make statements. With data errors you have a program that is syntactically sound but fails because the data values do not fit the program as it was written. With logic errors you have a program that runs, and data that fits, but the result is wrong because the program gave the wrong instructions to the computer."

For denne masteroppgåva er ikkje definisjonen av termen "feil" i seg sjølv så viktig. Viktigare er at ein forstår årsaken til at "eit eller anna ikkje fungerer som det skal".

8.7.3 Parvis testing som testteknikk for design av utval av testtilfelle

Metadatafila som Arkade prosesserer har mange inndata-parametrar.

Parvis testing [129] er ein testteknikk for å designe utval av testtilfelle. Teknikken byggjer på at erfaring ofte viser at dei fleste feil skuldast interaksjon mellom to faktorar. Testtilfelle generert ved hjelp av teknikken parvis testing dekkar alle kombinasjonar av to variablar samtidig.

Dustin [130] sin artikkel forklarar på ein pedagogisk god måte teorigrunnlaget for partesting. Ho kallar den "A method for deriving a suitable set of test cases", og rasjonaliserer den slik:

"It's a laudable goal; to conduct thorougly exhaustive test of a system. In the real world, however, it's generally not possible, feasible, or cost effective to test a system using all the possible variations and combinations of test parameter inputs".

Fordelen med denne testteknikken er at den reduserer sterkt behovet for testtilfelle utan at det går vesentleg utover dekningsgraden og sannsynlegheiten for å oppdaga feil. Cohen et. al [131] refererer til at forsøk med teknikken i Telcordia Tehnologies har medført sterk reduksjon av testkostnader:

"The authors describe an application in which the method reduced test plan development from one month to less than a week. In several experiments, the method demonstrated good code coverage and fault detection ability".

Også Kuhn [132] har eit optimistisk syn på å bruka parvis testing, og antydar at ein kan oppnå gode testresultat med teknikken:

"Teams seeking to maximize testing thoroughness given tight time or resource constraints, and which currently rely on manual test case selection methods, should consider pairwise testing. When more time is available or more thorough testing is required, t-way testing for t > 2 is better. Practitioners who require very high quality software will find that covering arrays for higher-strength combinations can detect many hard-to-find faults, and variability among detection rates appears to decrease as t increases. Sophisticated new combinatorial testing algorithms packaged in user-friendly tools are now available to enable thorough testing with a manageable number of test cases and at lower cost, and make it practical for testers to develop empirical results on applications of this promising test method."

"With the Web server application, for example, roughly 40 percent of the failures were caused by a single value, such as a file name exceeding a certain length; another 30 percent were triggered by the interaction of two parameters; and a cumulative total of almost 90 percent were triggered by three or fewer parameters. While not conclusive, these results suggest that combinatorial methods can achieve a high level of thoroughness in software testing."

Parvis testing er ein formell designteknikk, som byggjer på eit matematisk teoretisk fundament [131]:

"In the combinatorial design approach, the tester generates tests that cover all pairwise, triple, or n-way combinations of test parameters specified in formal test requirements. Covering all pairwise combinations means that for any two parameters p1 and p2 and any valid values v1 for p1 and v2 for p2, there is a test in which p1 has the value v1 and p2 has the value v2."

Cohen et. al skriv at parvis testing er veleigna til å designa utval av testtilfelle på grunnlag av kravspesifikasjonen:

"The first and perhaps most important step in defining test requirements is to identify the test parameters. In general, the parameters for unit testing are low-level entities, such as screen fields or message components, to which the tester must assign values. The system documentation usually makes clear what the unit testing parameters should be. The way to define test parameters for system testing, however, is less obvious. The key to defining these parameters is to model the system's functionality, not its interface. This reduces the model's complexity and focuses the model on the system aspects important to testing."

Ein får eit inntrykk av i kva for storleiksorden effektiviseringspotensialet kan vera [131]:

"We based the test requirements on the system's user manual. The test requirements for the final release had a total of 75 parameters with 1029 possible test combinations. The combinatorial design approach required only 28 tests to cover all pairwise parameter combinations for the 75 test parameters."

Det er ikkje nødvendigvis slik at fleire parametrar fører til behov for fleire testtilfelle [133]:

"The AETG input had one relation and modeled the configuration commands and the attenuator. The input for the first release had 61 fields; 29 fields with two values, 17 with three values, and 15 with four values. This gives a total of 229 x 317 x 415 = 7.4 x 10^{25} different combinations. The input for the second release had 75 fields; 35 fields with two values, 39 with three values, and 1 with four values. This give a total of 235 x 339 x 4 = 5.5 x 10^{29} different combinations. The AETG system generated 41 pair-wise tests for the first release and 28 pair-wise tests for the second. Even though the second release had many more combinations, pair-wise coverage required fewer tests. This illustrates the logarithmic growth properties of the AETG method. Even though the second release had six more fields with two

values and 22 more fields with three values, it required fewer tests because it has 14 fewer fields with four values."

Bach og Schroeder [87] er stort sett samde med andre artiklar om at parvis testing har mykje bra ved seg. Men av dei artiklane eg har lese, merkar denne artikkelen seg ut ved å vera kritisk også:

> *"This then is the pairwise testing story. See how nicely it fits together:*
>
> *1) Pairwise testing protects against pairwise bugs*
>
> *2) while dramatically reducing the number tests to perform,*
>
> *3) which is especially cool because pairwise bugs represent the majority of combinatoric bugs,*
>
> *4) and such bugs are a lot more likely to happen than ones that only happen with more variables.*
>
> *5) Plus, the availability of tools means you no longer need to create these tests by hand.*
>
> *Critical thinking and empirical analysis requires us to change the story:*
>
> *1) Pairwise testing* ***might find some*** *pairwise bugs*
>
> *2) while dramatically reducing the number tests to perform,* ***compared to testing all combinations, but not necessarily compared to testing just the combinations that matter****.*
>
> *3) which is especially cool because pairwise bugs* ***might*** *represent the majority of combinatoric bugs,* ***or might not, depending on the actual dependencies among variables in the product****.*
>
> *4) and* ***some*** *such bugs are more likely to happen than ones that only happen with more variables,* ***or less likely to happen, because user inputs are not randomly distributed****.*
>
> *5) Plus, you no longer need to create these tests by hand,* ***except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results****.*
>
> *The new story may not be as marketable as the original, but it considerably more accurate. Sorting it out, there are at least seven factors that strongly influence the outcome of your pairwise testing:*
>
> *1) The actual interdependencies among variables in the product under test.*

2) The probability of any given combination of variables occurring in the field.

3) The severity of any given problem that may be triggered by a particular combination of variables.

4) The particular variables you decide to combine.

5) The particular values of each variable you decide to test with.

6) The combinations of values you actually test.

7) Your ability to detect a problem if it occurs. "

8.7.4 Prinsippskisse for å illustrera effekten av parvis testing

Prinsippet for parvis testing er at ein kombinasjon av to variabelverdiar frå to variablar skal berre forekomma i eitt testtilfelle. Verdiar innanfor kvar testvariabel er gjensidig utelukkande for kvarandre for eit testtilfelle (i ei nedtrekksgardin kan ein berre velja ein verdi).

Hensikten med parvis testing er å få ein erfaringsbasert god reduksjon i talet på testtilfelle utan at det skal gå vesentleg utover kvaliteten på testinga. Altså ein metode for å effektivisera testinga av ein applikasjon.

I to eksempel under viser eg effekten av parvis testing.

Eksempel 1

Den første tabellen i eksempel 1 viser alle mogelege kombinasjonar av variabelverdiar.

Den andre tabellen i eksempel 1 viser dei kombinasjonane det blir redusert til når ein veljer parvis testing som testdesign-prinsipp.

Variabel	S_ds_A		S_ds_B		S_ds_C	
Variabelverdi	S-ds_AA	S-ds_AB	S-ds_BA	S-ds_BB	S-ds_CA	S-ds_CB
Testtilfelle 1	X		X		X	
Testtilfelle 2	X		X			X
Testtilfelle 3	X			X	X	
Testtilfelle 4	X			X		X
Testtilfelle 5		X	X		X	
Testtilfelle 6		X	X			X
Testtilfelle 7		X		X	X	
Testtilfelle 8		X		X		X

Tabell 11: Testdesign ved kartesisk produkt som prinsipp

Variabel	S_ds_A		S_ds_B		S_ds_C	
Variabelverdi	S-ds_AA	S-ds_AB	S-ds_BA	S-ds_BB	S-ds_CA	S-ds_CB
Testtilfelle 1	X		X		X	
Testtilfelle 2	X			X		X
Testtilfelle 3		X	X		X	
Testtilfelle 4		X		X		X

Tabell 12: Testdesign ved parvis testing som prinsipp

Eksempel 2

I dette eksempelet er talet på variabelverdiar for variabelen "S_ds_C" utvida frå 2 til 3. Formålet med eksempelet er både å visa kor lite som skal til for å auka talet på testtilfelle med 33% (frå 8 til 12) samt å visa kor håplaus oppgåve det vil bli å laga testdesign manuelt når kompleksiteten aukar.

Variabel	S_ds_A		S_ds_B		S_ds_C		
Variabelverdi	S-ds_AA	S-ds_AB	S-ds_BA	S-ds_BB	S-ds_CA	S-ds_CB	S-ds_CC
Testtilfelle 1	X		X		X		
Testtilfelle 2	X		X			X	
Testtilfelle 3	X		X				X
Testtilfelle 4	X			X	X		
Testtilfelle 5	X			X		X	
Testtilfelle 6	X			X			X
Testtilfelle 7		X	X		X		
Testtilfelle 8		X	X			X	
Testtilfelle 9		X	X				X
Testtilfelle 10		X		X	X		
Testtilfelle 11		X		X		X	
Testtilfelle 12		X		X			X

Tabell 13: Testdesign ved kartesisk produkt som prinsipp ved minimal auke i kompleksitet

Variabel	S_ds_A		S_ds_B		S_ds_C		
Variabelverdi	S-ds_AA	S-ds_AB	S-ds_BA	S-ds_BB	S-ds_CA	S-ds_CB	S-ds_CC
Testtilfelle 1	?	?	?	?	?	?	?
Testtilfelle 2	?	?	?	?	?	?	?
Testtilfelle 3	?	?	?	?	?	?	?
Testtilfelle 4	?	?	?	?	?	?	?
Testtilfelle 5	?	?	?	?	?	?	?
Testtilfelle 6	?	?	?	?	?	?	?

Tabell 14: Testdesign ved parvis testing som prinsipp ved minimal auke i kompleksitet

Alle spørsmålsteikna i tabell 13 illustrerer at berre ved å gå frå eksempel 1 til eksempel 2 tek det til å bli vanskeleg å laga eit design for parvis testing manult. Eit naturleg spørsmål er kva ein gjer då. I punkt 8.7.6 til slutt i dette kapittel har eg reflektert over muligheiter for implementering av testteknikk for parvis testing.

8.7.5 Grunnlagsskisse for design for testtilfelle ved dynamisk testing av Arkade

Det mest enkle design av inndata til Arkade framgår av følgjande tabell:

Program	**Nivå 1**
Arkade	Structure
	Processes

Tabell 15: Design av inndata frå meta-metadatafila til og med nivå 1.

Når vi legg til nivå 2, blir bildet meir komplisert:

Program	**Nivå 1**	**Nivå 2**
Arkade	Structure	Dataset
		File
		Recordtype
		Field
		Codelevel

	Processes	Dataset
		File
		Recordtype
		Field

Tabell 16: Design av inndata frå meta-metadatafila til og med nivå 2

For å få eit realistisk inntrykk av talet på variablar og variabelverdiar ved eit design for testtilfelle for å testa Arkade, må vi utvida til nivå 3. Dette visest i vedlegg 5. Vedlegg 5 viser at på det nederste nivået (nivå 3) er det 193 testattributter (variabelverdiar) fordelt på 61 variablar. Det vil sei i gjennomsnitt 3-4 verdiar per inndata-variabel. Viss vi berre for å gjera det enkelt seier at det er 61 variablar kvar med 3 verdiar, blir det med kartesisk produkt som design 3^{61} = 1,27173E+29 testtilfelle. Med 4 verdiar per inndata-variabel blir det 4^{61} = 5,3169E+36 testtilfelle.

8.7.6 Implementering av testteknikken parvis testing

Punkt 8.3.4 og 8.3.5 har illustrert behovet for parvis testing. Eg har gjort sonderingar om kva som finst av programvare tilgjengeleg for dette formål, og har funne programma Allpairs, PICT og CASEMAKER.

Allpairs

Programmet AllPairs [134] blir distribuert som GNU General Public Licence [135]. Det skal visstnok kunna lastast ned og takast i bruk gratis. Men eg har ikkje fått til å bruka dette, brukarrettleiinga er svært snau.

PICT

Programmet PICT er eit alternativ [136]:

"PICT has been in use at Microsoft Corporation since 2000. It was designed with usability, flexibility, and speed in mind, and it employs a simple (yet effective) core generation algorithm that has separate preparation and generation phases. This flexibility allowed for the implementation of several features of practical importance. PICT gives testers a lot of control over the way in which tests are generated. Additionally, it raises the level of modeling abstraction, and it makes pairwise generation both convenient and usable".

PICT står for "Pairwise Independent Combinatorial Testing". Eg lykkast ikkje å lasta ned programmet med Opera eller Safari nettlesar, men med Internet Explorer gjekk det greit. Men eg fekk ikkje til å brukar det. Ein brukarmanual som følgde med ved nedlasting er vanskeleg å bruka. Programmet har ikkje eit grafisk brukargrensesnitt, men er kommandolinje-basert.

CASEMAKER

Programmet CaseMaker kan også brukast til andre testteknikkar enn parvis testing [137]:

"The implemented techniques are: Equivalence Partitioning, Boundary Check, Error Guessing, Decision Tables and Pairwise Testing".

Produktet kostar bortimot 1000 USD. Eg har ikkje brukt dette programmet heller.

8.8 Testing av kombinasjonar av inndata i metadatafila

Mens vi i del-kapittel 8.4 såg på testteknikkar som var mest aktuelle for einheitstesting, er testinga i dette del-kapittelet mest aktuell for systemtesting, akseptansetesting og regresjonstesting. Dette er svartboks testing (vedlegg 1).

Testing i dette del-kapittel dreiar seg om dynamisk testing. Det er denne delen av testinga som i størst grad kan gje Arkivverket svar på om Arkade-programmet er til å stola på. Det må imidlertid vera klart at det generelt er slik at tillit til programvare uansett er noko som må byggjast opp over tid gjennom regelmessig bruk og vedlikehold. Om programmet er til å stola på, er på sidelinja av problemstillingane, men sidan masteroppgåva er skriven i samarbeid med Riksarkivet, er det naturleg at Riksarkivet som interessent ønskjer ein indikasjon på om programmet som ein stoppa utviklinga av for 5 år sidan er brukande.

Eit anna bidrag av dette del-kapittelet vil vera å gje ein indikasjon på om den utvikla testteknikken for kvalitetssikring og effektivisering av gjennomgangen av SAS-loggen fungerer tilfredsstillande.

8.8.1 Utval av testtilfelle

Opphavleg var planen at parvis testing skulle brukast til design av utval av testtilfelle. Som punkt 8.7.6 framfor viser, var arbeidet med implementeringen av testteknikken mislykka. Då overtok i praksis testteknikken utforskande testing denne rolla.

Tabellen under viser kva eg spesielt søkte å testa med dei ulike testtilfella. Denne tabellen var ikkje ferdig planlagt på førehand, men vart til undervegs med det praktiske testarbeidet. Det er karakteristisk for utforskande testing at teknikken tek små steg av gangen, og let neste steg byggja på det/dei forutgåande stega. Mi erfaring var at dette fungerte svært bra. For eksempel var det å la utdata-datafiler og utdata-metadatafiler frå eit testtilfelle vera inndata datafiler og inndata metadatafiler i påfølgjande testtilfelle ein svært effektiv måte å generera testtilfelle på. Dermed kunne meir tid brukast på den kreative delen med å laga ulike kombinasjonar av inndata i metadatafila.

Testtilfelle #	**Kva spesielt søker ein å testa?**
1	Ingen inndata. Testar om programmet greier endra frå norsk til engelsk språkdrakt
2	Testar om Arkade greier å lesa inn ei lita feilfri tekstfil. Fila sitt format er FIXED og postskiljeteikn er CRLF.
3	Innlesing av ei enkel tekstfil på 4 postar og 4 observasjonar. Konverterer den til filformat XML.
4	Les inn inndata-datafila frå testtilfelle 3 og konverterer fila frå teiknsettet ISO-8859-1

	til EBCDIC. Åpnar utdata-fila med FileViewer [138] og les den som EBCDIC.
5	Les inn inndata-datafila frå testtilfelle 3 og konverterer den frå teiknsettet ISO-8859-1 til ISO-8859-4.
6	Les inn utdata datafila frå testtilfelle 5, og konverter den tilbake frå teiknsettet ISO-8859-4 til ISO-8859-1. Undersøker om utdata-datafila frå dette testtilfellet er identisk med inndata-datafila frå testtilfelle 5.
7	Innlesing av to små tekstfiler med format FIXED + CRLF. Legg på ein konverteringsprosess på eit felt i kvar av filene for å sjå om konvertering frå karakterfelt til numerisk felt går OK.
8	Innlesing av dei same to teksfilene som i testtilfelle 7. Den eine fila blir konvertert til DELIM filformat mens den andre blir konvertert til XML filformat.
9	Innlesing av dei same to tekstfilene som i testtilfelle 8. Same konverteringsprosessar som i testtilfelle 8. I tillegg konverterast postskiljeteiknet på den eine fila som konverteast til DELIM frå CRLF til NO.
10	Innlesing av utdata-fila konvertert til DELIM og postskiljeteikn NO i testtilfelle 9. Fila konverterast til XML.
11	Innlesing av utdata-fila konvertert til DELIM og postskiljeteikn NO i testtilfelle 9. Fila konverterast til FIXED og postskiljeteikn CRLF.
12	Brukar dei to filene som vart utdata i testtilfelle 8, som inndata. Den datafila som vart konvertert til DELIM, blir no konvertert til XLM. Den datafila som vart konvertert til XML, blir no konvertert til DELIM.
13	Brukar dei to datafilene som vart utdata i testtilfelle 12, som inndata. Konverterer begge to tilbake til FIXED + CRLF. Testar om utdata datafilene frå dette testtilfellet er identisk med inndata-datafilene i testtilfelle 8.
14	Les inn 8 datafiler frå eit reelt deponert fagsystem Riksarkivet har motteke. Det er 8 filer med FIXED filformat og NO som postskilje. Tal postar i filene varierer frå 3000 til 35000. Tal på variablar er cirka 60. Postlengde varierer mellom 300 og 400 teikn.
15	Les inn dei same datafilene som i testtilfelle 14. Konverterer 4 av datafilen til XML filformat. Konverterer dei 4 andre datafilene til DELIM filformat med CRLF som postskiljeteikn.
16	Brukar dei 8 datafilene som vart utdata i testtilfelle 15, som inndata. Konverterer alle filene tilbake til filformat og teiknsett dei hadde som inndata for testtilfelle 15. Undersøker om utdata-filene frå testtilfelle 16 er identiske med inndata frå testtilfelle 15.
17	Endrar postlengden for ein av postane i inndata-datafila for testtilfelle 3. Undersøker om og eventuelt korleis Arkade reagerer på denne datafeilen.
18	Les inn ei datafil med FIXED og CRLF som postskiljeteikn. Lar metadatafila fortelja Arkade at inndata-datafila manglar postskiljeteikn (NO). Undersøker om og eventuelt korleis Arkade reagerer på denne mismatchen mellom datafil og metadatafil.

19	Les inn ei lita tekstfil med eit karakterfeltet POSTNR av datatypen STRING. Legg på ein prosess i metadatafila der eg aukar feltbreidden frå 4 til 8 for dette feltet. Legg også på ein prosess som konverterer datatypen GARDSNR frå INTEGER til STRING og aukar feltbreidden til 10 posisjonar.
20	Tek utgangspunkt i testtilfelle 7, men slettar den eine datafila som metadatafila fortel Arkade at skal lesast inn. Undersøker korleis Arkade reagerer på at ei datfil manglar.
21	Tek utgangspunkt i testtilfelle 7, og let metadatafila fortelja Arkade at eit felt må ha ein eintydig verdi. I den eine av filene har eg lagt inn duplikater i to postar, i den andre fila let eg feltverdien i ein av postane vera blank. Undersøker korleis Arkade reagerer på dette.
22	Innlesing av ei fil i FIXED format med CRLF. Konvertering av fila til DELIM med CRLF.
23	Innlesing av ei fil i DELIM filformat med CRLF. Konvertering av fila til FIXED med CRLF.

Tabell 17: Oversikt over utvalte testtilfelle ved dynamisk testing av Arkade

8.8.2 Meir detaljar om dei enkelte testtilfella

Testtilfelle 01

Det lykkast ikkje å få endra frå norsk til engelsk språkversjon ved å endra verdien på den globale makrovariabelen for språkdrakt.

Testtilfelle 02

Arkade greier fint å lesa fila inn i eit datasett i SAS. Det er ikkje lagt på konverteringsprosessar via metadatafila. Likevel har Arkade generert ei utdata-fil lik inndata-fila. Eg går ut ifrå dette er funksjonalitet ein bevisst har valt å ha med, sjølv om eg ikkje forstår grunnen. Startposisjonane til felta er endra i utdata-fila i forhold til inndata-fila, ved at blanke mellomrom mellom felta er borte. Det er uvisst om det også er tilsikta eller ikkje for dei krav som systemet er laga på grunnlag av. Eg har ikkje funne noko verken i systemdokumentasjonen [139] eller i kommentarar i kjeldekoden som seier noko om dette.

Testtilfelle 03

Både innlesing av datafila til eit datasett i SAS samt konvertering til ei XML-datafil går greit.

Testtilfelle 04

Både innlesing av datafila til eit datasett i SAS samt konvertering til ei fil med EBCDIC teiknsett går greit.

Testtilfelle 05

Både innlesing av datafila til eit datasett i SAS samt konvertering til ei fil med ISO-8859-4 teiknsett går greit.

Testtilfelle 06

Både innlesing av datafila til eit datasett i SAS samt konvertering tilbake til ei fil med ISO-8859-1 teiknsett går greit.

Testtilfelle 07

Arkade genererer 2 nye datafiler og ny metadatafil for dei nygenererte datafilene. Problemet ser ut til å vera at blanke teikn (mellom dei ulike felt) ikkje blir tekne vare på. Mellomrom mellom felta blir trunkerte ved konvertering frå STRING til INTEGER. Dermed får dei nye felta nye startposisjonar på kvar record. Verken i brukarmanualen eller systemdokumentasjonen kan eg finna som krav at konvertering av datatype for eit felt frå STRING til INTEGER ikkje skal føra til endringar i startposisjonen på felta i den konverterte fila. Den nye metadatafila som blir generert, viser dei nye startposisjonane for dei konverterte filene.

Eg testa ut med ulike dataverdiar i metadataelementet <mask>. Viss det er blankt, blir utfyllande blanke til slutt i det siste feltet på recorden tekne vare på, slik at CRLF kjem på same posisjon for alle postane. Dersom <mask> gis verdien '8.', vil eventuelle blanke bak talverdien bli trunkert, med den følgje at den konverterte fila ikkje får fast postlengde. Dette fenomenet er altså berre aktuelt å ta hensyn til når det er det siste feltet i posten som skal ha konvertert datatype.

Viss det er eit poeng at mellomrom mellom feltverdiar skal takast vare på i den konverterte fila, ser det ikkje ut som Arkade duger til det per idag.

Ein måte å omgå "problemet" på er å lesa inn frå dei originale filene med INTEGER i stadan for med STING . Men viss det er eit poeng at numeriske dataverdiar skal lagrast høgrejustert på den fysiske fila, ser eg ikkje at Arkade har funksjonalitet for det.

Testtilfelle 08

Både innlesing av datafilene til datasett i SAS samt konvertering til ei fil med DELIM format og XML format går greit.

Testtilfelle 09

I utdata-fila der postskiljeteiknet vart konvertert frå CRLF til NO, har Arkade greidd det.

Testtilfelle 10

Innlesing av utdata-fila konvertert til DELIM og postskiljeteikn NO i testtilfelle 9. Fila konverterast til XML. Arkade "går i heng" på dette testtilfellet. SAS krev bortimot 100% av all prosessorkraft på maskina. SAS må tvangsavsluttast ved å drepa systemprosessen via Windows Task Manager. Det er ikkje generert noko datafil eller metadatafil for konverteringa. Det blir ikkje skriven nokon rapport som ekstrakt for SAS-loggen, men sjølve loggfila blir teken vare på. Den kan brukast til å finna ut kvar det har gått gale i programmet.

Konklusjonen er at det ikkje går an å konvertera noko fil i DELIM format med NO som postskiljeteikn (altså ikkje noko postskiljeteikn) til XML-format.

Dette er ikkje noko rart, for det er uråd for Arkade å vita kvar postskiljet er i ei DELIM-fil (kommaseparert fil) utan eksplisitt postskilje, sidan det heller ikkje er skiljeteikn etter siste felt eller framfor det første i postane.

Testtilfelle 11

Innlesing av utdata-fila konvertert til DELIM og postskiljeteikn NO i testtilfelle 9. Fila konverterast til FIXED og postskiljeteikn CRLF. Arkade "går i heng" på dette testtilfellet også. SAS krev mykje ressursar av maskina, 60-70% av total CPU. Etter å ha stått og "hengt" ei stund, tvangsavsluttast SAS ved å drepa systemprosessen via Windows Task Manager. Det er generert ei datafil på nokre millionar postar, men dei er tomme for innhold. Einast er CRLF som er på fast posisjon. Ny metadatafil blir ikkje generert. Det blir ikkje skriven nokon rapport som ekstrakt for SAS-loggen, men sjølve loggfila blir teken vare på. Den kan brukast til å finna ut kvar det har gått gale i programmet.

Konklusjonen er iallfall at det går ikkje å konvertera ei kommaseparert fil utan postskiljeteikn til FIXED format med CRLF.

Forklaringa er gjeve til slutt under testtilfelle 10 over.

Testtilfelle 24 viser iallfall at det går fint å konvertera ei fil med DELIM og CRLF til FIXED og CRLF.

Testtilfelle 12

Konverteringa frå DELIM til XML var vellykka.

Men konverteringa frå XML til DELIM er ikkje heilt vellykka. Den siste posten i fila blir duplisert i utdata, men utan at den avsluttast med CRLF. Rapporten med ekstrakt av viktige meldingar frå SAS-loggen viser 10 WARNING-meldingar. Undersøking av loggfila viser konteksten for desse meldingane. SAS reagerer på bruk av variablar som ikkje finst i datasett. Feilmeldinga oppstår i den delen av programmet som bereknar feltlengder. Ein raskt titt på koden kan tyda på at lengdeutrekningsdelen er av dei mest kompliserte delane i programmet. Det bør først ryddast opp i programmeringsstilen som er brukt, deretter er det håp om å spora seg fram til feilens årsak.

Testtilfelle 13

Ved konvertering frå XML til FIXED eller frå DELIM til FIXED, går det ikkje an å styra startposisjonen til felta med elementet <startpos> i metadatafila.

Både datasett og utdata-datafil blir laga, men dei er tomme.

Ved generering av metadatafila i Arkadukt [140], gis det valideringsmelding i Arkadukt om at det ikkje har nokon hensikt å setja startposisjon, sluttposisjon eller fast feltlengde (elementet ft_fixlength) når formatet til fila som feltet tilhøyrer, ikkje er fast (FIXED).

For å undersøkja loggen nærmare, køyrest programmet på nytt utan parsing av loggen. Loggen går til default fil, og kan undersøkjast i logviewet i SAS. Av konteksten i SAS-loggen går det fram at alle dei 10 WARNING-meldingane kjem ved utføring av lengdeutrekningsprogrammet i Arkade. Det skjer når makroen %GET_LENG blir kalla i programfila GET_KONV.sas. Makroen %GET_LENG er definert i programfila MG_P_GEN.sas.

Testtilfelle 14

Ingen ERROR eller WARNING i ekstraktet frå SAS-loggen. Innlesinga i datasett går greit, som vist i tabellen under.

I motsetning til testtilfelle 2, blir det ikkje generert utdata-datafiler.

Filnamn	Talet på postar	Innlest i SAS
F2	36136	36136
F5	35555	35555
F8	35467	35467
F11	35589	35589
F14	35479	35479
F17	35409	35409
F20	35461	35461
F23	2481	2481

Tabell 18: Resultat ved testing av testtilfelle 14

Testtilfelle 15

Ingen ERROR eller WARNING i ekstraktet frå SAS-loggen. Kontroll av talet på postar i utdata-datafiler for DELIM og talet på postar for alle filene stemmer overeins. Talet på postar i XML-filene er ikkje undersøkt (går ikkje med FileViewer).

Filnamn	Talet på postar	Innlest i SAS	Utdata DELIM
F2	36136	36136	
F5	35555	35555	
F8	35467	35467	
F11	35589	35589	
F14	35479	35479	35479
F17	35409	35409	35409
F20	35461	35461	35461
F23	2481	2481	2481

Tabell 19: Resultat ved testing av testtilfelle 15

Testtilfelle 16

Eg får feilmelding frå Arkadukt ved innlesing av metadatafila som vart utdata i testtilfelle 15. Feilmeldinga lyder:

"The field is too small to accept the amount of data you attempt to add. Try inserting or pasting less data. (-2147217833 – ImporterXMLNode)".

Eg redigerer metadatafila manuelt. Det var ein del feil ved denne konverteringen. Analyserapporten frå SAS-loggen viser WARNING-meldingar av same type som for testtilfelle 13. Det tyder på at feilen er den same. Nemleg at det er noko feil med lengdeutrekninga i Arkade.

Filnamn	Talet på postar i inndata datafila	Talet på observasjonar i SAS datasett	Talet på postar i utdata datafila (FIXED/CRLF)	Kommentarar
F2	36136	36137	36137	Siste og nest siste post er duplikater. Ikkje CRLF for siste post på datafila.
F5	35555	35556		Fila vart ikkje skriven til disk etter konvertering.
F8	35467	35468	35468	Siste og nest siste post er duplikater. Ikkje CRLF for siste post på datafila.
F11	35589	35589	??	Ikkje nok minne på maskina til at FileViewer greidde å lasta fila inn i minnet. I SAS er siste og nest siste post duplikater. Den siste posten i datafila mangla som observasjon i datasettet.
F14	35479	35479	35479	
F17	35409	35409	35409	
F20	35461	35457	35457	4 postar for lite innlest i SAS. Dei to siste postane er duplikater. Ikkje CRLF for siste post på datafila.
F23	2481	2481	2481	Ikkje CRLF for siste post på datafila.

Tabell 20: Resultat ved testing av testtilfelle 16

Testtilfelle 17

Postlengden for post nr. 2 kortast ned med nokre få teikn. Resultatet er at SAS les inn 3 postar i stadan for fire. Post nr. 3 blir ikkje lest inn. Når SAS kjem til CRLF for post nr. 2, les SAS forbi postsluttmerket inntil starten av neste linje. Det ser vi ved at verdien av det siste feltet på linje 2 har det første teiknet på linje 3 som sitt siste teikn. Det kjem ingen WARNING- eller ERROR-melding frå SAS-loggen. Aktivering av sjekk for metadataelementet <reclength> i Arkade på filnivå og metadataelementet <analysis> på datasettnivå ned til filnivå, gjev heller ingen melding om dette fenomenet i analyserapportane som Arkade genererer.

Testtilfelle 18

Datasettet får 3 postar i stadan for dei 4 det skulle hatt. Den siste posten i datafila blir ikkje lest inn i datasettet. Første felt i andre observasjon i datasettet inneheld teikn som er uleselege, kanskje CRLF frå første posten? I tredje posten er det blankt innhold i første feltet. Einaste måten å oppdaga ein feil som dette på, er ved inspeksjon av datasettet i SAS.

Testtilfelle 19

Fungerer heilt greit.

Testtilfelle 20

Metadatafila fortel Arkade at to filer skal lesast inn i SAS, men berre ei av filene finst der metadatafila fortel at to filer skal finnast.

Arkade les inn den fila som finst i eit datasett som ser fullstendig ut ved inspeksjon etterpå. Det er ingen WARNING- eller ERROR-melding frå SAS-loggen.

Det ser heller ikkje ut som Arkade er utstyrt med ein prosess for å sjekka dette forholdet. Det bør vera eit standard kontrollpunkt for testprogram i Arkivverket at det blir testa for om ei deponering faktisk inneheld det som metadataene seier at den inneheld. Erfaringsmessig er det ofte mismatch her. Det kan vera at ei deponering inneheld enten færre eller fleire datafiler enn det metadataene seier den inneheld.

Testtilfelle 21

Metadatafila fortel Arkade at eit felt skal vera unikt i ein post. I den eine fila er det dublettar i dette feltet, i den andre fila er verdien blank i ein post. Arkade rapporterer avviket når kontrollprosessen for UNIQUE er lagt på. SAS melder ingenting via loggen.

Testtilfelle 22

Innlesing av ei fil i FIXED format med CRLF og konvertering av fila til DELIM med CRLF går greit.

Testtilfelle 23

Innlesing av ei fil i DELIM filformat med CRLF og konvertering av fila til FIXED med CRLF går greit.

8.8.3 Testresultat

Tabellen under viser oversikt over feil og manglar ved ulike testtilfelle ved dynamisk testing av Arkade.

Testtilfelle #	Tal ERROR-meldingar i SAS-loggen	Tal WARNING-meldingar i SAS-loggen	Tal feil påvist	Feil nr.
1	0	0	1	1
2	0	0	0	-

3	0	0	0	-
4	0	0	0	-
5	0	0	0	-
6	0	0	0	-
7	0	0	0	-
8	0	0	0	-
9	0	0	0	-
10	0	0	1	2
11	0	0	1	3
12	0	10	1	4
13	0	10	1	5
14	0	0	0	-
15	0	0	0	-
16	0	37	1	6
17	0	0	1	7
18	0	0	1	8
19	0	0	0	-
20	0	0	1	9
21	0	0	0	-
22	0	0	0	-
23	0	0	0	-

Tabell 21: Feil og feilmeldingar ved ulike testtilfelle ved testing av Arkade

Tabellen under viser oversikt over dei påviste feil ved testing av Arkade.

Feil #	**Feilens karakter og årsak**	**Feilen retta?**
1	Arkade skiftar ikkje til engelsk språkdrakt. Ukjent årsak. Lite alvorleg feil.	Nei
2	Det går ikkje an å konvertera ei fil med DELIM format utan postskiljeteikn til XML format. Arkade "går i heng". Alvorleg	Nei

	feil. Burde vore lagt inn feilhåndteringslogikk i programkoden for å avslutta på kontrollert vis med meldingar til SAS-loggen.	
3	Arkade "går i heng". Alvorleg feil. Burde vore lagt inn feilhåndteringslogikk i programkoden for å avslutta på kontrollert vis med meldingar til SAS-loggen.	Nei
4	Feil i programkoden for utrekning av feltlengder.	Nei
5	Feil i programkoden for utrekning av feltlengder.	Nei
6	Feil i programkoden for utrekning av feltlengder.	Nei
7	Arkade gjev ikkje melding om datafeil, at postlengden for ei inndata datafil med FIXED postlengde er endra.	Nei
8	Arkade gjev ikkje melding om at det er misforhold mellom datafila og metadatafila for ei fil med FIXED og CRLF der metadatafila oppgjev at postskiljeteiknet manglar.	Nei
9	Arkade gjev ikkje melding om at ei datafila som metadatafila fortel Arakde at finst, ikkje finst.	Nei

Tabell 22: Påviste feil ved dynamisk testing av Arkade

8.8.4 Vurdering av testresultat

Det kan tyda på at dei feil som er funne, er av tre typar:

- For dårleg feilhåndtering i programlogikken (feil nr. 2 og 3).
- Feil i lengdeutreknings-programmet (utrekning av feltlengder). Dette gjeld feil nr. 4, 5 og 6.
- For dårleg validering av samsvar mellom metadatafila og datafilene. Dette går både på struktur (feil nr. 7 og 8) og talet på filer (feil nr. 9).

14 testtilfelle køyrer feilfritt og 9 feilar.

Det er vanskeleg å evaluera dette med at utdata-filene ikkje har dei same startposisjonar på filer av typen FIXED som inndata-filene, men fjernar blanke mellomrom mellom felt, fordi evalueringa av programmet må relaterast til kva kravet er meint å vera.

8.8.5 Vurdering av testteknikk

Eit relevant spørsmål er om dei fleste av dei 23 testtilfella er gjennomført ved å køyra Arkade i seg sjølv eller om det er testapplikasjonen eg har bygd opp som blir køyrt. Svaret er "ja" på begge deler.

Kalla på dei aktuelle programfilene skjer frå testapplikasjonen. Når det gjeld fila AUTOEXEC.sas, er det ein modifisert kopi av den originale programfila i Arkade som blir køyrt. Den originale kunne ikkje brukast, fordi den var knytt direkte mot brukargrensesnittet

til Arkade. Likeeins har eg sjalta ut dei SCL-programfilene som er i Arkade, og som direkte gjer knytningen mellom applikasjonen og brukargrensesnittet. Det er altså ikkje brukargrensesnittet i Arkade-applikasjonen som tek imot inndata-parametrane som fortel kvar datafilene og metadatafila ligg eller kvar utdata-filer skal plasserast. Det er også testapplikasjonen som kallar programfilene med alle makroane eg har utvikla i samband med testinga av Arkade. For eksempel alle makroane for analyse av SAS-loggen. Hovudlinjene i dette er forklart i punkt 8.2.

Men svaret er også "ja" på spørsmålet om det er Arkade-applikasjonen i seg sjølv som har køyrt. Arkade har 75 programfiler, og i hovudsak er det desse som eksekverast når eit testtilfelle køyrer. Viktige unntak er filene METH_INI.sas, METH_DIC.sas, METH_GEN.sas og METH_RUN.sas.

Dette er 4 viktige filer som i Arkade vert kalla direkte av SCL-programfilene. For å bli uavhengig av grensesnittet, måtte eg oppretta kopiar av desse programfilene i testapplikasjonen.

Det vesentlege poenget er at hensikten med å ha ein testapplikasjon, er å få kontroll over kva for kode i Arkade som skal køyrast når. Ved trykking på knappar i eit grensesnitt får eg ingen reell kontroll med dette. Ikkje den nødvendige kontroll til å grava meg ned på eit tilstrekkeleg detaljert nivå til å finna ut eksakt kvar i koden ein feil er. Eit grensesnitt er til hjelp for den som skal bruka ein applikasjon. Men det kan like gjerne "stå i vegen" for ein som skal testa ein applikasjon. All innomhus testing i Riksarkivet som skjedde av Arkade då den var under utvikling, skjedde ved at grensesnittet "stod i vegen" for den som testa. Slikt kan det blir både frustrasjon og misforståingar av.

Både loggfila frå SAS-loggen og analyserapportane (word-fil) er tekne vare på for alle testtilfella.

Dei fleste av testtilfella vart køyrt enkeltvis. Men eg køyrde også fleire sekvensielt, eller køyrde det enkelte testtilfellet fleire gonger, og i dei tilfella vart både loggfilene og analyserapportane tekne vare på alle saman (ingen overskriving), noko som skuldast makroen %SASLOG_NY_LOGGFIL_M_TIDSSTEMPEL som opprettar namn på desse filene med tidsstempel (månad-dag-time-sekund) som namneidentifikator.

Ei praktisk utfordring eg støytte på var at koden i Arkade for utskrift av analyserapportar, brukar prosedyren PROC PRINTTO i SAS. Bruk av denne prosedyren opphevar skrivinga av SAS-loggen til den loggfila eg styrer loggen til ved parsing av loggfila. I dei tilfella der eg testa på om Arkade rapporterte forhold til sine eigne analyserapportar, kunne eg ikkje skriva heile loggen til den parsa loggfila i same køyringa. Eg løyste problemet ved å gjera køyringa av PROC PRINTTO i Arkade betinga av ein global makrovariabel, som eg let styra programlogikk som eg utvikla og implementerte dei stadane i Arkade der det vart brukt PROC PRINTTO. Måten å finna desse stadane på var ved søking på strengen 'proc printto' i Windows utforskar.

Alle testtilfella vart køyrt i produksjonsmodus. Grunnen var at det oppstod ikkje feil som eg ikkje forstod med å bruka produksjonsmodus. For testtilfella 10 og 11 der Arkade "gjekk i heng" ville eg ikkje hatt nytte av debuggingmodus i første omgang. Eg vil tru at testteknikken for å spora loggmeldingar til korrekt programfil vil kunna brukast for å avgrensa søket etter feil # 2 og 3. Denne testteknikken inneheld 3 makroar som er dokumenterte i vedlegg 15.

To eksempler på testing som ikkje er omtalt tidlegare i masteroppgåva, er testing av testapplikasjonen og testing av testteknikken for analyse av SAS-loggen. Dette er gjort bevisst, for ikkje å ta fokus bort frå Arkade-applikasjonen. Men både testapplikasjonen og makroane som utgjer testteknikken for analyse av SAS-loggen er i seg sjølv ei kjelde til feil, og om ein feil stammar frå ein eller fleire testteknikkar eller frå Arkade-applikasjonen er heilt vesentleg. Herved er det sagt; både testapplikasjonen og testteknikken for analyse av SAS-loggen har også vore testa. Ikkje minst har testteknikken for analyse av SAS-loggen vore nyttig ved testing av testapplikasjonen. Eitt av designvala for testapplikasjonen var "sentralisert vedlikehold og kompilering av koden". Det er mi subjektive overtyding at dette har ført til at testapplikasjonen og makroane for analyse av SAS-loggen har vorte grundig gjennomtesta. Det vart ikkje er funne ERROR eller WARNING-meldingar i loggen for køyringa av desse ved den dynamiske testinga av Arkade. Begge vart eksekverte for kvart testtilfelle.

9 Diskusjon

Masteroppgåva har søkt å gje svar på både generelle og spesielle problemstillingar. Dei generelle problemstillingane er diskutert i eit tidlegare kapittel.

Dette kapittelet inneheld:

- Diskusjon om nytten av testteknikken utforskande testing.
- Diskusjon av resultat av arbeidet i forhold til dei spesielle problemstillingane.
- Diskusjon om det er likheitspunkt eller parallellar mellom resultata for dei generelle og spesielle problemstillingane.
- Diskusjon om Arkade er brukande.

9.1 Diskusjon om nytten av testteknikken utforskande testing

Julie Gardiner hadde utforskande testing som ein av sine favoritt-teknikkar for systemtesting og akseptansetesting. Dynamisk testing av dei 23 testtilfella kan også karakteriserast som systemtesting / akseptansetesting.

Sjølv om eg har hatt gode erfaringar med utforskande testing, kan eg ikkje verken støtta eller ikkje støtta testar B sine erfaringar om at utforskande testing finn andre feil enn dei spesifiserte testtilfella påviser. Iallfall ikkje direkte, men indirekte kan jo testteknikken ha spela ei rolle ved å velja andre testteknikkar som så i sin tur har vist seg å vera komplementære til dei feil som er funne ved den dynamiske testinga ved hjelp av testtilfella.

Jorgensen sin kontekst for utforskande testing, ein applikasjon utvikla av nokon andre som skal debuggast / vedlikeholdast og med ein utydeleg kravspesifikasjon, passar godt på den kontekst Arkade er i.

For utvikling av testapplikasjonen hadde eg mykje nytte av utforskande testing. Dette arbeidet bestod både av utvikling og testing. Det var ein iterativ prosess med utvikling og testing av testapplikasjonen. Det blir læring, kodeutvikling og eksekvering av koden parallellt. Då er vi ved kjernen av Bach sin definisjon av utforskande testing. I underkapittel 8.2 har eg dokumentert utforskande testing si rolle ved utvikling av testapplikasjonen.

Bach beskriv i del-kapittel 7.1 fem grunnelement i utforskande testing: Test design, nøyaktig observasjon, kritisk tenkning, kreative idear og bruk av hjelperessursar.

1. Utforskande testing har vist seg god for design av utval av testtilfelle. Vi kan også snakka om design i forstand av samarbeid mellom dei makrovariablar, formater og makroar som utgjer testteknikken for analyse av SAS-loggen.

2. Nøyaktige observasjonar har vore nødvendig både av inndata og utdata (talet på postar) for dei store filene som vart testa i nokre av testtilfella.

3. Kritisk tenkning har det vore mykje av, eg har hatt eit høgt bevisstheitsnivå på at feil like gjerne kan stamma frå eigenutvikla kode som frå Arkade når eg brukar eigenutvika kode for å testa Arkade.

4. Når det gjeld det fjerde grunnelementet – kreative idear – synest eg ikkje heuristikkane har vore så relevante.

5. Men for det femte grunnelementet – hjelperessursar – er det full klaff. Ein god utforskande testar byggjer faglege nettverk og erfaringsdatabasar. Eg ser i denne samanheng på brukargruppene for SAS som eit fagleg nettverk. Ein rask titt på litteraturlista viser at desse har spela ei viktig rolle for utvikling av testteknikken for analyse av SAS-loggen for eksempel. Dette har eg også nemnd i del-kapittel 8.1.

I del-kapittel 7.2 hevdar Bach at det generelt vil passa å bruka utforskande testing i einkvar situasjon der det ikkje er opplagt kva den neste testen bør vera, eller der testinga ikkje er fullstendig diktert på førehand. Det passar godt som kontekst for masterprosjektet, der fint lite eller ingenting har vore fastlagt på forhånd.

I same del-kapittel gjev Delorio uttrykk for at det å gjera erfaringar undervegs og ta hensyn til desse i problemløysinga vidare, er ei god beskriving av utforskande testing. Det var akkurat slik eg opplevde det ved design av dei 23 testtilfella. Dei vart til undervegs.

Heilt til slutt i del-kapittel 7.2 legg Delorio stor vekt på at utforskande testing vektlegg forståing av konteksten feil opptrer i. Det er viktigare å finna kjelda til feilen enn feilen sin manifestasjon. Dette oppfattar eg som ein annan måte å sei det på at feilen sin natur er viktigare å forstå enn å få feilen klassifisert etter ein eller annan grupperingsvariabel. Dette er det same eg sjølv hevdar i punkt 8.7.1 (Evaluering av tidegare testplanar for Arkade) og punkt 8.7.2 (Fokus på feila sin natur framfor kategorisering av feil).

Både Julie Gardiner og Zylbermann (del-kapittel 7.3) skiljer mellom ad hoc testing og utforskande testing. Aguss og Johnsen (punkt 7.3) ser på testteknikkar langs aksen formelle / mindre formelle testar. Artikkelen deira set likheitsteikn mellom det å vera ein formell testteknikk og det å byggja tillit til eit program. Men dei innrømmer at uformelle teknikkar kan vera fullt så effektive til å finna feil. Det blir litt ulogiske for meg viss ein testteknikk som finn feil ikkje eignar seg så godt for å byggja tillit.

9.2 Diskusjon av dei spesielle problemstillingane

Dei spesielle problemstillingane var:

- Korleis bør Arkivverket sitt SAS-program Arkade testast?
- Kor lett testbart er dette programmet?
- Korleis kan det eventuelt gjerast meir testbart?

Som utgangspunkt valte eg testteknikken utforskande testing. Testapplikasjonen vart utvikla med denne tilnærminga. Det var mykje lesing av teori og utprøving av praktiske kodeeksempel for å komma fram til noko som dekka dei behov eg meinte var relevante.

Eit vesentleg element ved utforskande testing, er å tilpassa seg ulike kontekstar og kunna tilpassa tilnærming og testteknikk. Å bruka utforskande testing tyder i praksis å bruka dei testteknikkar som akkurat i øyeblikket fortonar seg mest hensiktsmessig. For eksempel brukte eg mykje testteknikken kodegjennomgang for dei testteknikkane eg foreslår implementert for å gjera programmet lettare testbart:

- Testing av returkode.
- Testing ved bruk av kode som avsluttar eksekveringa.
- Testing av struktur og vedlikeholdbarheit i koden.
- Testing av gyldigheitsområde for markovariablar.
- Testing av eksekverbar kode frå ein makro ved hjelp av ein annan makro.
- Testing ved å la globale makrovariablar erstatta kommentar-teikn i kjeldekoden.
- Testing av korleis Arkade tek vare på evidens til bruk for å bli testa.
- Testing av bruken av makroparametrar i Arkade.

Viss eg skal prioritera fire områder der Arkade har størst potensiale for å bli lettare testbart, kan eg foreslå:

- Programmeringsstilen bør bli betre. Det vil gjera det lettare for andre å setja seg inn i programlogikken i koden. Det vil gjera det enklare og tryggare å gjera endringar i koden. Det vil også gjera det enklare å oppdaga muligheiter for ytterlegare modularingsering av koden.
- Globale makrovariablar bør erstatta bortkommentering av kode for testinga av programmet. Slik programmet er no, ligg mykje testkode ubrukt fordi det er for tungvint å bruka den.
- Implementering av større robustheit for feilhåndtering. Her kan det vera aktuelt både med testing av returkode og bruk av kode som avsluttar eksekveringa med gode feilmeldingar til SAS-loggen.
- Einkvar makro bør bli definert i si eiga, dedikerte programfil. Det vil gjera det enklare å finna definisjonen av ein makro. For ein del makroar er det slik, men ikkje for alle. Konvensjonen som er brukt er at programfila heiter det same som makroen. Det er ein god konvensjon, fordi den lettar søket etter fila der makroen er definert.

Gjennom testinga av dei 23 testtilfella har testapplikasjonen og testteknikken for analyse av SAS-loggen vist seg svært nyttige. Når det er syntaksfeil i koden, er det grunn til å rekna med ERROR- eller WARNING-meldingar i loggen. Feil nr. 4, 5 og 6 gjev alle mange WARNING-meldingar i loggen, og det er grunn til å tru at dette er feiltypar som skuldast syntaksfeil eller semantiske feil i programmet.

Feil nr. 7, 8 og 9 kan tyda på at er logiske feil. Det er ingen feilmeldingar til loggen, men resultatet blir annleis enn forventa. Til denne feiltype kan ein truleg også rekna feil nr. 1.

Feil nr. 2 og 3 er vanskelege å debugga fordi Arkade "går i heng", timeglasset for ein pågåande prosessering blir ikkje borte på skjermen, datafila veks kontinurerleg på disken og programmet må tvangsavsluttast. SAS-loggen gjev ingen rapport, men loggfila blir teken vare på for så langt som inntil programmet blir tvangsavslutta. Her trur eg dei utvikla makroane for sporing av loggmeldingar til korrekt programfil, vil vera til hjelp (makroane %SASLOG_MSG_START_EXEC_FILE og %SASLOG_MSG_END_EXEC_FILE). For at Arkade ikkje skal bli hengande i ei tilsynelatande evig løkke og generera ei utdata-datafil av aukande storleik, trur eg testing av returkode og testing ved bruk av kode som avsluttar eksekveringa ville vore nyttig å implementert.

Nyttig ville det også vore å testa ved å la globale makrovariablar erstatta kommentar-teikn i kjeldekoden. Også justeringar i koden slik at Arkade tek vare på evidens til bruk for å bli testa. I praksis vil det å gjera koden lettare testbar og det å testa programmet bli to sider av same sak.

Det praktiske testarbeidet i testtilfella har brukt testteknikken svartboks testing. Vi kan også kalla det for systemtesting eller akseptansetesting. Det er den testteknikken som tidlegare vart brukt ved testing av Arkade under utvikling av programmet. Ut ifrå dei feil som er avdekka, meinar eg det var rett vurdert å velja testtilfelle og svartboks testing i denne samanheng. Dette har ført til at einheitstesting, så som testing av den enkelte makro, har vorte prioritert bort av tidshensyn. Med avgrensa kjennskap til programmet sin indre struktur, er det mi vurdering at for å få testa på einheitsnivå krevest meir tid enn det eit masterprosjekt har. Fokus for masteroppgåva er også eit litt anna enn å få eit dataprogram til å fungera. Testinga har vist at den utvikla testteknikken med parsing og uttrekk av viktige meldingar frå SAS-loggen har fungert godt, og det har vore intensjonen. Like fullt har eg utvikla ein makro for å testa eksekverbar kode frå ein makro, og eg trur også den vil komma til nytte ved einheitstesting av programmet i tida framover.

Dei fleste av testtilfella er gjennomført med små datafiler. Det synest vera tilstekkeleg for å få fram vesentlege feil i programmet viss ein ser bort ifrå eigenskapar som har med yting å gjera, og som ikkje er uvesentelege for den praktiske bruk programmet er laga for. Mange filer i Arkivverket er store, ofte mange hundre tusen postar. Den reelle deponeringa som vart testa gjekk rimeleg raskt å prosessera.

Hensikten med testapplikasjonen var mellom anna å gjera Arkade lettare testbart, ved samtidig å frikobla seg frå det grafiske grensesnittet. Det har vist seg å fungera greit.

Eg erfarte at det var ikkje berre var disponibel tid som var begrensning for testteknikken parvis testing for utvikling av testdesign. Det er ein del kombinasjonar av inndata som er irrelevante. Programmet Arkadukt gjev melding om det ved parsing av metadatafila. Men det er ingenting ved registrering av metadataelement i metadatafila (ved bruk av Arkadukt) som forhindrar brukaren i å registera irrelevante elementverdiar. For eksempel er postlengde og feltlengde irrelevante å registera når filformatet på inndata-fila er XML. Dette vil redusera talet på testtilfelle ein del ved parvis testing. Det vil også setja krav til at eit verktøy som skal brukast til å laga testdesign for denne testteknikken, kan ta hensyn til slike forhold.

9.3 Diskusjon av dei generelle og spesielle problemstillingane samla sett

Det er fleire punkt det går an å trekka parallellar mellom resultat som har framkomme i arbeidet med dei generelle og spesielle problemstillingane. I det følgjande påpeikar eg ein del slike.

Utføringen av ein testteknikk

Artikkel 5 trekker fram at det like gjerne kan vera utføringa av ein testteknikk som testteknikken i seg sjølv som medverkar til om feil blir funne. Skriving av %PUT-setningar i koden for å få fram verdiar av makrovariablar til loggen ved køyring av eit program, er ein vanleg testteknikk i SAS-program. Ein kan sjå på utføringa av denne teknikken som om den blir utført slik den i dag er implementert i Arkade eller alternativt slik eg foreslår den utført ved å la globale makrovariablar erstatta bortkommentering av kode for testing av programmet.

Testverktøyet

Testar A har erfart at testverktøy og testteknikk i aukande grad har vorte nemnd i same åndedrag i løpet av dei siste åra. Det same kan ein sei om den testteknikken eg har utvikla og brukt for å kvalitetssikra gjennomgangen av SAS-loggen. Ein testteknikk som kan brukast til å testa såvel Arkade som testapplikasjonen som skal testa Arkade.

Den enkelte feil sin natur

Artiklane 1, 2 og 5 meinar det meir er den enkelte feil sin natur enn den type/kategori feilen måtte vera klassifisert under, som avgjer om feilen er lett å oppdaga eller ikkje. Feil nr. 2 og 3 som fører til at programmet "går i heng" utan meldingar til loggen som er til spesiell hjelp, er eksempler på feil som er vanskelege å finna ut av. Feil nr. 4, 5 og 6 som gjeld feil i programkoden for utrekning av feltlengder, vil eg tru er litt lettare å spora opp fordi loggen her gjev WARNING-meldingar som det er grunn til å tru har samanheng med feilen.

Testmiljø

Verdien av eit velfungerande teknisk testmiljø er poengtert av Lloyd Roden, Julie Gardiner, Telenor og Statens Pensjonskasse. Dei erfaringane eg har gjort underbyggjer dette. Dei val eg gjorde for utviklinga av testmiljøet, har gjort at denne koden er gjennomtesta og av god kvalitet.

Innstilling til å finna feil

Eg har tidlegare diskutert ulike tilnærmingar (skular) til testing på eit overordna plan. Eit relevant spørsmål å stilla seg kan vera kor mykje hjelp det er i slike overordna tilnærmingar når ein sit med eit kodeproblem framfor seg. Dilorio meinar at holdningar hos den som testar er like viktig som testverktøy [82]:

"Knowing the debugging tools and how to use them is all fine and well. Remember, however, that behavior and attitude are at least as important as tools when fixing a program."

14 testtilfelle køyrer feilfritt og 9 feilar. Eg har hatt som innstilling å finna feil (destruktiv testing), for eksempel i testtilfelle 10 og 11 (feil # 2 og 3) der Arkade "går i heng".

Eit råd eg synest eg har hatt god bruk for, er frå Dilorio som tilrår at ein les programkoden (testteknikken kodegjennomgang) som om ein var ein kompilator [82]:

"Examine the program as if you were the SAS compiler – read the statements in the order in which they were executed and describe the function of each statement as stated in the statement's syntax. This requires you to make the sometimes difficult distinction between your knowledge of the program's context and your knowledge of SAS syntax".

Kontekst

Eg valde testteknikken utforskande testing mellom anna på grunn av at den så tydeleg er kontekst-orientert. Telenor vektlegg å velja form for statisk testing basert på kontekst (punkt 4.3). Roden tilrår å vurdera testestimering på bakgrunn av kontekst (punkt 4.1). Både for forhold relevante for dei generelle og spesielle problemstillingane er kontekst viktig. I punkt 6.5 (Korleis bør Arkade testast i lys av dette kapittel) refererast at både Roden, Bach & Schroeder og Whitmill gjev uttrykk for at "Beste Praksis" ikkje er eit allmenngyldig fenomen, men er kontekstavhengig.

Testerfaring og testkompetanse

Artiklane nr. 1 og 5 påpeikar at testerfaring og testkompetanse er like viktig som testteknikken i seg sjølv. Dette blir støtta av testarane A og B, samt Lloyd Roden. Eg kan ikkje skilta med formell kompetanse i testing i form av sertifisering. Men eg har nok bygd opp litt erfaring gjennom bruk av SAS. Dilorio meinar at til tross for mange hjelpemidler, er den viktigaste hjelpa likevel den kompetanse ein sjølv byggjer opp gjennom eigen erfaring [82]:

"In the long run, you are your best resource. Nothing compares to developing a body of knowledge built on application and industry-specific experience. As you build more complex programs, break them, and repair them, the speed and reliability of the repairs will increase. Most important, the number of errors will decrease as you write savvier, more bullet-proof code."

Dilorio føreset at eikvar problemløysing bør brukast til å bringa seg sjølv og sin eigen erfaring eit steg vidare. Generelt hevdar han at det er viktig å ha eit åpent sinn, og ikkje vera for sikker i starten på kva årsaken til ein feil er:

"The flip side of relying on your experience is that it is, indeed, limited to being only your experience. It's important to accept the idea that the bug may have arisen from factors beyond those in your personal history."

Utviklingsmetodikk

Testar B har erfart at lettvektsmetodikkar finn feil meir effektivt enn meir tradisjonelle fossefallsmetodikkar. Lloyd Roden påpeikar at det er eit ukjent behov for tal på iterasjonar i lettvektsmetodikkar, eller i utvikling generelt for den del.

Testapplikasjonen for Arkade er utvikla inkrementelt. Dilorio meinar at program bør testast/debuggast og utviklast inkrementelt [82]:

"Develop and Fix Programs Incrementally. Even simple programs are best developed in small steps. This becomes even more important when you are writing logically complex or long DATA steps or when you are using new PROCs. The reason is simple – if you take a functioning program, make a single change, and see the program fail, the failure can be attributed to the change just made. Make five changes, however, and the identification of the errant code can become problematic. Note that "single change" does not necessarily mean a single statement or parameter. Rather, it is a small, logically related group of statements or options."

Dilorio foreslår å bruka eit kodenummer i kommentarar alle stader i koden (kan dreia seg om mange programfiler) som blir berørt av ei endring i koden. Ved søking på den tekstlege strengen som den unike kodeverdien representerer, vil ein lett finna fram til alle stader i programmet som endringa gjeld for. Dette kan vera nyttig i neste omgang, når det blir aktuelt å endra denne delen av koden på nytt. Denne teknikken brukte eg då eg kommenterte bort PROC PRINTTO i koden ved betingselsestesting avhengig av verdien til ein global markovariabel.

Dilorio [141] gjev gode råd om utviklingsmetodikken for debugging av SAS-program:

- Bruk prosedyrer og verktøyet datasteg debugger [142].
- Bruk PUT og LIST setningar.
- Lag gode diagnostiske meldingar når du programmerer.
- Bruk loggen og utdata aktivt.
- Bruk systemopsjonar for å kontrollera utdata.

Mangelfull bruk av systemopsjonar er noko eg har påpeikt for Arkade. Det å laga gode diagnostiske meldingar når ein programmerer, kunne like gjerne vore sagt slik at ein bør brukar returkode og programmera for kontrollerte avslutningar av programmet. Å bruka loggen aktivt har eg gjort ved utviklinga av testteknikken for å auka kvaliteten på gjennomgangen av loggen etter køyring. For dei største køyringane (testtilfella 14, 15 og 16) er loggfila på 7-8000 linjer.

9.4 Er Arkade brukande?

Eit spørsmål som naturleg reiser seg, spesielt frå Arkivverket si side, vil vera om Arkade er brukande eller ikkje. Som påpeikt i punkt 1.3 innleiingsvis, vart utviklinga av programmet stansa av Riksarkivet i 2004, fordi det angiveleg skulle vera mange feil i programmet, likevel utan at Riksarkivet dokumenterte overfor leverandøren eksempler på konkrete inndata som førte til feilsituasjonar. Masteroppgåva har skaffa litt betre kjennskap til ein del av feila. Det

er utvikla eit testmiljø (ein testapplikasjon) og testteknikkar som delvis er implementert. I sum borgar dette for at testinga kan bli betre enn før.

Termen ”brukande” er relativ, ikkje absolutt. Spørsmål om Arkade er brukande eller ikkje, kan ha like mykje med forventningane hos den som spør å gjera som det har med programmet å gjera. Eller reflektera at den som spør ikkje er bevisst nok på at termen er relativ. Spørsmålet om Arkade er brukande, er for viktig til å bli besvart med ”Ja” eller ”Nei”. Det kan vera freistande å referera Ratcliffe [103] som gjev til kjenne ein realistisk og jordnær ambisjon for debugging og for kvalitet på programvare generelt:

”If we cannot eliminate human error we are unlikely to be able to eliminate the creation of bugs in computer programs. However, the inevitability of bugs in all but the smallest of programs should not deter us from attempting to minimise the number of bugs; and those bugs that we do produce should be easy to investigate and resolve”.

I kapittel 1 såg vi at Bjørnerstedt [7] har ei tilnærming som imøtegår det generelle inntrykket at det er så mange utviklingsprosjekt som blir mislykka:

“Det er en grov forenkling å sette likhetstegn mellom et budsjett som overskrides og det å mislykkes. Det finnes mange eksempler på prosjekter som går langt over budsjett på grunn av at ambisjonen (scope) har økt. Et prosjekt som går over budsjett med 50 % men som også leverer 50 % mer verdi enn planlagt kan vel ikke kalles mislykket?”

Avisa Computerworld har den 30.04.2010 ein artikkel under overskrifta “Grønt lys for skandaleprosjekt”. I 2007 stoppa Statens vegvesen eit av Norges største offentlege it-prosjekt, utviklinga av systemet Autosys, men no er det lagt ut på anbod på nytt. Grunnen til at det vart stoppa i 2007, er ifølgje fungerande prosjektleiar Steinar Rask at ein sist ikkje hadde eit tydeleg styringsdokument over kva som skulle gjerast undervegs. For utviklingsprosjektet for Arkade var det også slik at kravspesifikasjonen var fråværande/manglande i utgangspunktet. Avisa Computerworld refererer forskar Magne Jørgensen, som på generelt grunnlag påpeikar at offentlege prosjekt ofte får uforholdsmessig mykje kritikk [143]:

“En av årsakene kan være at det offentlige oftere er en dårlig kunde. Det skydes blant annet at det er mindre it-kompetanse. Derfor er det også oftere budsjettoverskridelser og problemer med styringen”.

Jørgensen blir referert av Computerworld på eit fenomen han kallar informasjonsasymmetri [143]:

“Leverandørene vet mye mer om hva de utvikler enn kunden. Har man lite kunnskap blir det man avtaler ofte basert på urealistisk lav pris. Det medfører en del komplikasjoner rundt leveransen”.

Det som bør telja mest i vurderinga om vellykka eller mislykka prosjekt er ifølgje avisa Computerworld sin referanse til Jørgensen [143]:

“Det er ikke bra at it-prosjekter til stadighet overskrider budsjettene, men enda viktigere er at investeringen har vært riktig og at utviklingsarbeidet er gjennomført effektivt.

Det blir ofte til at mediene vektlegger det som er lettest målbart og ikke det som er viktig i prosjektet".

Det Jørgensen kallar ein "riktig investering" tolkar eg som likt med det Bjørnerstedt omtalar som "systemets verdi" og då uavhengig av om det lykkast på kravspesifikasjon, tid og budsjett [7]:

"Hvis man snur på spørsmålet og ser på hva som kjennetegner et vellykket IT prosjekt så vil mange si at det må levere et system i henhold til spesifikasjon på tid og budsjett. Problemet er bare det at det finnes mange eksempler på prosjekt som oppnår akkurat dette men der systemet likevel har liten verdi. Forklaringen til dette paradokset er at spesifikasjonen er dårlig. Andre ganger har behovene endret seg underveis".

Ifølgje Bjørnerstedt kan det vera ei vanleg misforståing å tru at berre kravspesifikasjonen er fyldig nok, så er den dermed god [7]:

"En fremstående bedriftsleder sa en gang at han var sikker på at 50 % av reklamebudsjettet var bortkastet. Problemet var bare at han ikke visste hvilken halvdel det var. Når man utvikler et IT system er det heller ikke alltid mulig å vite på forhånd hvilke funksjoner som vil bli brukt. Det som dog er sikkert er at dagens prosesser (særlig i offentlig sektor) med omfattende kravspesifikasjoner og deretter kontrakt basert på oppfyllelsen av disse ikke er veien å gå. Resultatet blir at kunden legger inn alt i spesifikasjonen som muligens kan bli etterspurt. Leverandøren har ikke heller noe incitament for å finne ut hva som egentlig vil bli brukt. Man vinner kontrakt gjennom å tilby det kunden ber om, ikke gjennom å tilby det kunden har behov for".

Bjørnerstedt meinar løysinga på dette dilemmaet er å bruka smidige metodar der produktet sin karakter er slik at ein kan gjera det:

"I senere år har interessen for det som kalles smidige metoder (agile development) stadig vokst. Et av de store poengene med disse metodene er nettopp å dreie fokus fra å utvikle etter en spesifikasjon til å inkrementelt utvikle det som gir kunden mest verdi. Tanken er at hyppige leveranser skal gjøre kunden i stand til å justere sine mål og holde fokus på det som virkelig har verdi. En utfordring med å ta disse metodene i bruk er at mange kontrakter fortsatt binder både kunde og leverandør til en statisk spesifikasjon. Det er med andre ord fortsatt en lang vei å gå".

Smidig utvikling [2] er ein utviklingsmetodikk som mellom anna har som kjenneteikn sterk grad av brukarinvolvering. Ein av måtane å få det til på er ved parprogrammering [144].

"Pair programming is an agile software development technique in which two programmers work together at one work station. One types in code while the other reviews each line of code as it is typed in. The person typing is called the driver. The person reviewing the code is called the observer (or navigator). The two programmers switch roles frequently (possibly every 30 minutes or less).

While reviewing, the observer also considers the strategic direction of the work, coming up with ideas for improvements and likely future problems to address. This frees the driver to focus all of his or her attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide".

Dette er langt sterkare grad av brukarinvolvering enn den rolle tilsette i Riksarkivet hadde i prosjektet med utviklinga/testinga av Arkade. Men då snakkar vi like gjerne om Arkivverket sin bedriftskultur som vi snakkar om produktet Arkade eller systemutviklingsmetodikkar.

10 Konklusjonar

Dette kapittelet gjev ei samanfatning og avslutning av masteroppgåva. Del-kapittela er:

- Samanfatning av funn
- Kritikk av eige arbeid
- Bidrag til fagfeltet
- Vidare arbeid

10.1 Samanfatning av funn

Testteknikkar spelar ei rolle for kva for feil som blir funne, og i praksis opplever mange testarar at ulike testteknikkar er komplementære til kvarandre (dei finn ulike feil). Men det er også mange andre forhold som er viktige, og ofte er det uråd å fastslå forklaringsvariablane sin relative verdi.

Det er mange forhold som er identifisert som felles for testing og debugging for dei generelle og spesielle problemstillingane. Det indikerer at desse forholda er allmengyldige. Eksempler på slike forhold er:

1. Utføringen av ein testteknikk.
2. Testverktøyet.
3. Den enkelte feil sin natur.
4. Testmiljø.
5. Innstilling hos den som testar til å finna feil.
6. Testerfaring og testkompetanse.
7. Utviklingsmetodikk.
8. Programmeringsstil.
9. Konteksten si rolle.

Eg har valt testteknikken utforskande testing som hovudteknikk for å testa Arkade. Denne teknikken er aktuell når det gjeld vedlikehold av gammal kode, når det er dårleg kjent om krava er implementert i programkoden eller når rask læring er nødvendig

Testteknikkane kodegjennomgang erbrukt mykje. Testteknikken inspeksjon er også mykje brukt i samanhengen inspeksjon av datasett.

SAS-loggen har ein sentral plass ved testing og debugging av SAS-program. Ved hjelp av utforskande testing har eg utvikla ein testteknikk for å kvalitetssikra gjennomgangen av SAS-loggen.

Eg har utvikla ein testapplikasjon for å testa Arkade. Denne testapplikasjonen utgjer den vesentlege delen av testmiljøet.

For design av testtilfelle har eg argumentert for prinsippa som ligg til grunn for testteknikken parvis testing. Praktisk arbeid med testing har vist at den må tillempast litt på grunn av at alle kombinasjonar for alle variabelverdiar for alle inndata ikkje er relevante.

Viss eg skal prioritera fire områder der Arkade har størst potensiale for å bli lettare testbart, kan eg foreslå:

- Programmeringsstilen bør bli betre. Det vil gjera det lettare for andre å setja seg inn i programlogikken i koden. Det vil gjera det enklare og tryggare å gjera endringar i koden. Det vil også gjera det enklare å oppdaga muligheiter for ytterlegare modularingsering av koden. Sjå vedlegg 7.

- Globale makrovariablar bør erstatta bortkommentering av kode for testinga av programmet. Slik programmet er no, ligg mykje testkode ubrukt fordi det er for tungvint å bruka den. Sjå vedlegg 10.

- Implementering av større robustheit for feilhåndtering. Her kan det vera aktuelt både med testing av returkode og bruk av kode som avsluttar eksekveringa med gode feilmeldingar til SAS-loggen.

- Einkvar makro bør bli definert i si eiga, dedikerte programfil. Det vil gjera det enklare å finna definisjonen av ein makro. For ein del makroar er det slik, men ikkje for alle. Konvensjonen som er brukt er at programfila heiter det same som makroen. Det er ein god konvensjon.

På grunnlag av resultata som er dokumenterte i vedlegg 13, har eg funne at Arkade har følgjande forbetringspunkt for betre å utnytta innebygd funksjonalitet i Arkade for testing:

- Systemopsjonen SOURCE2 bør takast i bruk

- Systemopsjonen DATASTMTCHK bør setjast til ALLKEYWORDS

- Systemopsjonen MLOGIC bør enkelt kunna veljast framfor NOMLOGIC

- Systemopsjonen MPRINT bør flyttast frå fila MIXC_DSE.sas til AUTOEXEC.sas, men ei forklaring i kjeldekoden på bruken av den

- Systemopsjonen MLOGICNEST bør takast i bruk ved testing av Arkade

- Systemopsjonen MPRINTNEST bør takast i bruk ved testing av Arkade

- Systemopsjonen SYMBOLGEN bør enkelt kunna veljast ved testing av Arkade

Vedlegg 11 konkluderer med at Arkade tek ikkje godt nok vare på evidens for å bli testa, fordi datasett overskriv kvarandre, og slettar dermed spor bak seg.

23 testtilfelle representerer det praktiske testarbeidet som er gjort med Arkade. Det er funne 9 feil sålangt i programmet. Resultata tyder på at dei feil som er funne, er av tre typar:

- For dårleg feilhåndtering i programlogikken (feil nr. 2 og 3).
- Feil i lengdeutrekningsprogrammet (utrekning av feltlengder). Dette gjeld feil nr. 4, 5 og 6.
- For dårleg validering av samsvar mellom metadatafila og datafilene. Dette går både på struktur (feil nr. 7 og 8) og talet på filer (feil nr. 9).

10.2 Kritikk av eige arbeid

Dette del-kapittelet kan sjåast på som eit refleksjonsnotat etter avslutta arbeid med masteroppgåva.

Lang oppgåve på deltid – fordelar og ulemper

Arbeidet med masteroppgåva har pågått over ein 4-årsperiode, på deltid ved sida av å vera i 100% stilling på ordinær dagtid. Det er ikkje noko kritikkverdig i at eit mastergradsarbeid blir gjort under slike rammevilkår, men det er klart at det kunne hatt sine fordeler for arbeidsprosessen isolert sett om det kunne ha vore meir kontinuitet ved at eg hadde hatt høve til å driva med det på fulltid. Særleg i sluttfasen, når rapportskriving og dokumentasjon av utført arbeid kjem i tillegg til dynamisk testing og kodeutvikling. Spesielt for utvikling av programkode hadde det vore ein fordel å kunna arbeida utan hyppige avbrot for å dra på arbeid, fordi kodeutvikling er lett å komma utav, og det trengs tid til å setja seg inn att i koden slik situasjonen var før avbrotet.

Men på den andre sida – når arbeidet går over såpass lang tid totalt – får idear høve til å modnast, utprøvast og forkastast. Ikkje minst det å forkasta idear kan vera nyttig. Ein får eit godt inntrykk av kva det tyder i praksis å arbeida iterativt, når det blir tid nok til også å forkasta idear. Dette gjeld såvel design av masteroppgåva, val av problemstillingar, lesing av litteratur som den praktiske kodeutviklinga. Det har vore mykje litteratur å forholda seg til, og det er mykje litteratur som i tillegg til litteraturlista er lese og forkasta som irrelevant. For eigen kompetanse innan fagfeltet testing har masteroppgåva vore eit kraftig løft.

Design av masteroppgåva

Masteroppgåva var opphavleg tenkt gjennomført som eit eksperiment, men undervegs endra det seg til eit case-studium. Ideen om eksperiment var motivert ut ifrå tanken om at ulike testteknikkar kunne prøvast på ulike delar av programmet, for å finna ut om ein testteknikk var meir effektiv enn ein annan. *For det første* gjorde litteraturstudium det klarare og klarare at svaret på det spørsmålet var at det er såpass mange andre forhold enn testteknikken i seg sjølv som spelar ei rolle, at det ville vorte store metodiske problem med å holda tilstrekkeleg mange variablar konstante (intern validitet). *For det andre* vart det klart at med konteksten for masteroppgåva - ein student åleine - ville eit resultat, uansett kva det hadde vorte, fått metodiske problem med ekstern validitet (å vera representativt for eit større utvalg). Med auka kunnskap til fagfeltet testing undervegs i arbeidet, erkjente eg også at kvantitative undersøkingar av statistisk natur, ville vore nødvendige i eit eksperiment, og ikkje var eigna i denne samanhengen, og neppe ville ha relevans i industri-samanheng heller. *For det tredje*

ville det vera svært vanskeleg å vita for visst om det var råd å dela Arkade opp i uavhengige modular. Viss dei ikkje er uavhengige av kvarandre, kan resultat i ein modul tenkjast å influera på resultat i ein annan modul, og det skaper metodiske problem når resultat skal samanliknast. Ideen var å samanlikna om ein testteknikk, brukt på ein modul, var meir eller mindre effektiv for å finna feil enn ein annan teknikk brukt på ein annan modul. Det kan gjerne sjå overkommeleg ut når ein ser på ein designmodell av programmet, som systemdokumentasjonen inneheld figurar av. Men i praksis blir det vanskeleg. Det er mange inndata-parametrar i metadatafila som kan kombinerast med kvarandre på ulike måtar. Det er vanskeleg å ha sikre meiningar om alternative stiar gjennom programmet for ulike innparametrar, og dermed kan "modulgrenser" lett bli ein føresetnad som ikkje held i praksis. Det kan vera mange forhold som ikkje let seg fanga opp i forenkla designmodellar. Forsøket på oppdeling av programmet i modular ville føresett større grad av kvitboks testing (vedlegg 1, sjå også [45]) enn det kjennskapet til programmet gjev grunnlag for. Statisk testing av programmet har avdekka at koden først bør få ein betre programmeringsstil før ein kan intensivera einheitstestinga.

Teori

Svakheit ved å bruka ein metaanalyse [33] som referanse i artikkel 5 er at det lett kan bli slik at eg overlet ansvaret for evaluering og analyse til andre. Styrken er at dette også er erfaringar, og erfaringar som fortrinnsvis er akkumulert og aggregert på eit breidt grunnlag.

Teoretisk gjennomgang av studiar og metastudiar synest visa at Software Engineering i utgangspunktet har ein svak posisjon for å "bevisa" at studiar er homogene. Det er relativt få studiar det dreier seg om tross alt, dermed blir ikkje den statistiske styrken så stor. Miller [145] skriv:

"Can we find evidence in the studies to explain these potential differences? Considering the first potential partition, in their article Roper et al. statistically show that the performance of the detection techniques is dependent on the 'type' of faults encountered. This type of interaction has also been noted by several of the other studies, which report that the techniqus tend to find diffent types of defects, and hence benefit can be derived by combining the techniques. Although some evidence exists to an important effect, further exploration would be pointless, as we are not in a position to transform the outcome into moderator variables, due to the lack of definition of defect type. This inability to transform moderator explanations into variables is a major problem within Software Engineering, as we will be left in a position of being unable to partition the population into homogeneous subpopulations."

Miller er også skeptisk til om metastudiar kan brukast til så mykje i praksis [145]:

"Can meta-analysis be successfully applied to current Software Engineering experiments? The question is investigated by examining a series of experiments, which themselves investigate - which defect detection technique is best? Applying meta-analysis techniques to the Software Engineering data is relatively straightforward, but unfortunately the results are highly unstable, as the meta-analysis shows that the results are highly disparate and don't lead to a single reliable conclusion. The reason for this deficiency is the excessive variation within various components of the experiments."

Likevel, at noko ikkje er perfekt tyder ikkje at det ikkje er brukande. Kvar for seg kan ein sei at både eksperiment, metastudiar, strukturerte intervju og faglege foredrag som Software 2010 er utilstrekkelege, men tilsaman er sjansen større for at det kan trekkast ut brukbar essens frå dei.

Empiri

Dei testteknikkar eg har valt å fokusera på i masteroppgåva, gjev seg på ingen måte ut for å vera uttømmande eller fullstendige verken for Arkade-applikasjonen eller for debugging av SAS-program generelt. For eksempel har SAS ein innebygd funksjonalitet som kallast datasteg debugger [142]. Eg skulle gjerne ha utforska denne, men tida strakk ikkje til.

Sjølv om testteknikken utforskande testing vart valt i utgangspunktet, har eg også brukt mange andre teknikkar. Det er ikkje slik å forstå at det er noko hierarkisk forhold mellom testteknikkar. Parvis testing er for eksempel ikkje nokon "underteknikk" av utforskande testing. Om det ikkje skulle ha komme fram tydeleg nok i masteroppgåva så langt, skal det seiast her at testteknikkar ofte (oftast?) ikkje er å sjå på som ekskluderande alternativ til kvarandre, men som likestilte bidragsytarar.

Parvis testing blir mykje omtala i kapittel 8, men i praksis blir den ikkje brukt. Forsvaret mitt for å gje testteknikken såpass mykje plass, er som det står til slutt i dette kapittel; at arbeidet med masteroppgåva peikar framover, og at det er lagt "eit solid fundament" for testing og debugging av Arkade-applikasjonen. Parvis testing er eit *teoretisk* fundament for vidare testing utover masteroppgåva.

Del-kapittel 8.8 gjev ikkje direkte svar på ein einaste av problemstillingane i masteroppgåva. Eg har likevel presentert tre gode grunnar i innleiinga til del-kapittel 8.8 for å ta med denne dynamiske testinga [100] i oppgåva. Mens den dynamiske testing i kapittel 8 viser kva for feil eg har funne, er problemstillingane av ein slik karakter at dei krev analytiske svar.

10.3 Bidrag til fagfeltet

Generelt

Gjennomsyn av tidlegare leverte masteroppgåver ved Institutt for informatikk viser at det er relativt få som er innanfor fagfeltet software testing. I dag er det ikkje dedikerte kurs i programvaretesting ved Institutt for Informatikk ved UiO. Det er å håpa at denne masteroppgåva kan bidra til å heva eller synleggjera testfaget sin relevans og plass i eit slikt studium.

Eg er medlem i dataforeningens faggruppe for Software testing. Masteroppgåva vil bli presentert for styret i Faggruppa for Software Testing, som ei gjenyting mot at eg fekk lov å delta på Software 2010 til sterkt rabattert studentpris.

Faggruppa for Software Testing har eit ønskje om å løfte testfaget opp til å vera ein eigen profesjon/ karriereveg i Norge (arbeid mot studieinstitusjoner). Innanfor studiar til ingeniør-, data- og informatikkfag har testing ikkje hatt nokon sjølvstendig rolle inntil ganske nyleg, men vore undervist til ei viss grad integrert med andre fag. Sidan hausten 2008 har dataforeningen si faggruppe for software testing gjennomført testkurs på NTNU i Trondheim, samt fått i gang eit testfag på NITH i Oslo frå høsten 2009. Faggruppa [146] har stått

ansvarleg for undervisningsopplegget, og nesten heile styret har stilt opp som forelesere og utarbeida pensum og eksamensoppgaver for studentene.

Det gjenstår å sjå om denne masteroppgåva kan brukast i undervisningen deira, eller om den kan vera inspirasjon til andre studentar som ønskjer å skriva ei oppgåve til ein bachelor- eller mastergrad innanfor emnet programvaretesting.

Oppgåva trekker linjer og samanhenger mellom testing og debugging av programvare i ein generell og spesifikk kontekst. Oppgåva argumenterer for at kontekst er ein svært viktig variabel for val av testmetodar. Oppgåva søker å leita etter "Beste Praksisar", men er samtidig kritisk til dette uttrykket som eit allmenngyldig fenomen og har fleire konkrete eksempler på det både i den generelle og spesifikke delen.

Oppgåva argumenterer for at det gjev større meining og betre resultat i debuggingsprosessen viss ein fokuserer på dei einskilde feil og lærer å kjenna deira presentasjonsform enn å ha fokus på å gruppera feil etter ein eller annan grupperingsvariabel (alvorlegheitsgrad, opphav).

Arkivverket som interessent

Å få ei grundig evaluering av Arkade-applikasjonen er interessant for Riksarkivet (Arkivverket). Det kosta i si tid mykje pengar å få systemet utvikla, og resultata av testinga av systemet i denne masteroppgåva har vist at koden i programmet har mykje å bidra med av funksjonalitet for det vidare arbeidet med kvalitetskontroll av filer arkivdepotet mottek og konvertering mellom ulike filformat av elektronisk arkivmateriale, samt å ha ein rolle som eit nyttig verktøy i vidareforedling av arkivmateriale til ulike brukstjenester.

SAS-brukarar og SAS-utviklarar

På to hovudområder har masteroppgåva allmenn interesse utover Arkade-applikasjonen:

- utvikling av eit testmiljø (vedlegg 3) for å testa Arkade. Med små justeringar kan testapplikasjonen (Arkade_TestApp) tilpassast som testmiljø for å testa andre applikasjonar enn Arkade som er utvikla i SAS.

- testteknikken for analyse av SAS-loggen (vedlegg 14) kan brukast av einkvar som brukar SAS-systemet. Makroane og makrovariablane kan brukast direkte som dei er, eller vera inspirasjon for å tilpassa dei til sitt eige behov.

Det er også deler av testarbeidet som er gjort som SAS-utviklarar kan henta inspirasjon frå til å tilpassa eller gjenbruka som det er, for eksempel:

- testing av returkode (vedlegg 6)

- testing ved bruk av kode som avsluttar eksekveringa

- skilnaden på god og dårleg struktur av koden med tanke på vedlikeholdbarheit (vedlegg 7)

- testing av gyldigheitsområde for makrovariablar (vedlegg 8)

- einheitstesting av ein makro ved hjelp av ein annan (test)makro (vedlegg 9)

- utvikling som støttar testing ved å la globale makrovariablar erstatta kommentar-teikn i kjeldekoden
- eit bevisst forhold til å ta vare på evidens til bruk for seinare testing av koden (vedlegg 11)
- eit bevisst forhold til val av parametertyper ved utvikling av makroar (vedlegg 12)

10.4 Vidare arbeid

Ein blir aldri "ferdig" med å utvikla/vedlikeholda programvare i den forstand av ordet "ferdig" at all vidare utvikling kan frysast. Program vil alltid ha behov for å utviklast vidare for å møta nye behov. Viss ei "frysing" av kodeutvikling skjer, blir ein applikasjon gradvis meir og meir "legacy code" [86].

Arbeidet med masteroppgåva har prioritert å laga eit solid fundament for testing og debugging av Arkade-applikasjonen. Dette har ført til at det har ikkje vorte tid nok for ei testing av applikasjonen i tråd med det tilrådde designet (parvis testing) for utval av testtilfelle. Både framtidig bruk av applikasjonen og dedikert testing av den i henhold til testteknikken parvis testing kan vera aktuelt i framtida.

I det forutgåande punkt 9.4 blir utviklingsmetodikkar diskutert generelt. Det gis ingen klare råd, men påpeikast forhold som er relevante å ta hensyn til ved ved vidare utvikling/testing av Arkade eller forsåvidt for utvikling og testing av program generelt.

Litteraturliste

1. Brooks, F.P., *The Mythical Man-month*. 1995: Addison Wesley Lingman, Inc. 322.

2. Wikipedia. *Agile software development*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Agile_software_development#Agile_practices.

3. Wikipedia. *Debugging*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Debugging.

4. Wikipedia. *Software testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Software_testing.

5. Wikipedia. *Fault (technology)*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Fault_(technology).

6. Computerworld. *Derfor går it-prosjektene i vasken*. 2008 [cited 30.04.2010]; Available from: http://www.idg.no/computerworld/article110268.ece.

7. Bjørnerstedt, N. *Hva er et mislykket IT prosjekt?* 2008 [cited 30.04.2010]; Available from: http://www.leanway.no/?p=7.

8. Bjørnerstedt, N. *Det enkle er ofte det beste*. 2008 [cited 30.04.2010]; Available from: http://www.leanway.no/?p=9'%20rel='external%20nofollow.

9. Bentley, J.E. *Software Testing Fundamentals - Concepts, Roles and Terminology*. in *SAS Users Group International Proceedings*. 2005. Philadelphia, Pennsylvania: SAS Users Group International Proceedings (SUGI 30).

10. Wikipedia. *Kort historikk om programmeringsspråket SAS.* [cited 30.04.2010]; Available from: http://no.wikipedia.org/wiki/SAS_Institute.

11. Wikipedia. *SAS (software)*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/SAS_(software).

12. Eberhardt, P. *The SAS Data Step: Where Your Input Matters*. in *SAS Users Group International Proceedings*. 2005. Philadelphia, Pennsylvania: SAS Users Group International Proceedings (SUGI 30).

13. Howard, N. *How SAS Thinks or Why the DATA Step Does What It Does*. in *SAS Users Group International Proceedings*. 2003. Seattle, Washington: SAS Users Group International Proceedings (SUGI 28).

14. Howard, N. *Introduction to SAS Functions*. in *SAS Users Group International Proceedings*. 1999. Miami Beach, Florida: SAS Users Group International Proceedings (SUGI 24).

15. Wikipedia. *Interpreter (computing)*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Interpreter_(computing).

16. SAS_Institute. *Saving and Storing Frame SCL Programs*. [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/appdevgd/59506/HTML/default/a001005081.htm.

17. Carpenter, A.L., *Building and Using Macro Libraries*, in *The Pharmaceutical Industry SAS Users Group (PharmaSUG)*. 2002, The Pharmaceutical Industry SAS Users Group (PharmaSUG).

18. Koomen, T., et al., *TMap Next for result-driven testing*. 2006: UTN Publishers. 752.

19. Veenendaal, E.v. and H. Schaefer (2006) *Terminologi for test av programvare*. Oversettelse til norsk av Standard glossary of terms used in Software Testing Version 1.2 (June 2006) **Volume**, 63

20. Cooper, A. *Debugging Standard Macros*. in *PhUSE 2007*. 2007: GlaxoSmithKlinek, Harlow, UK.

21. Oslo, I.f.l.o.n.s.I.v.U.i. and N. Språkråd. *Bokmålsordboka og Nynorskordboka*. 2010 [cited 2010; Available from: http://www.dokpro.uio.no/ordboksoek.html.

22. Wikipedia. *Forklaring av den engelske termen "Failure"*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Failure.

23. Wikipedia. *Code review*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Code_review.

24. Wikipedia. *Inspection*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Inspection#Software_engineering.

25. Arkivverket. *Standarden ADDMML*. [cited 2010; Available from: http://www.arkivverket.no/arkivverket/Arkivbevaring/Elektronisk-arkivmateriale/Standarder.

26. Røykenes, K., *Metodetriangulering - Essay i vitenskapsteori*. 2006, Senter for Vitenskapsteori, Universitetet i Bergen. p. 18.

27. Wikipedia. *Exploratory testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Exploratory_testing.

28. Natalia, J. and V. Sira, *Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect?*, in *Empirical Methods and Studies in Software Engineering*. 2003, Springer Berlin / Heidelberg. p. 208-232.

29. Victor, R.B. and W.S. Richard, *Comparing the Effectiveness of Software Testing Strategies.* IEEE Trans. Softw. Eng., 1987. **13**(12): p. 1278-1296.

30. Roper, M., M. Wood, and J. Miller, *An empirical evaluation of defect detection techniques.* 1997. **39**(11): p. 763-775.

31. Andersson, C., et al. (2003) *An Experimental Evaluation of Inspection and Testing for Detection of Desing Faults*. **Volume**,

32. Laitenberger, O., *Studying the effects of code inspection and structural testing on software quality*. 1998: p. 237 - 246.

33. Runeson, P., et al., *What Do We Know about Defect Detection Methods?* Vol. 3. 2006.

34. Software Engineering Group, K.U., UK and U.o.D. Department of Computer Science, UK (2007) *Guidelines for performing Systematic Litarature Reviews in Software Engineering*. **Volume**, 65

35. Wikipedia. *System testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/System_testing.

36. Wikipedia. *Test automation*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Test_automation.

37. Wikipedia. *Testcase*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Testcase.

38. Wikipedia. *Non-functional testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Non-functional_testing.

39. Wikipedia. *Performance testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Performance_testing.

40. Wikipedia. *Unit testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Unit_testing.

41. Wikipedia. *Integration testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Integration_testing.

42. Wikipedia. *Destructive testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Destructive_testing.

43. Wikipedia. *Stress testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Stress_testing.

44. Kumar, A. and K. Graham, *Kan flere feil oppdages ved bruk av ulike testmetoder?* 2003, NITH / Toll- og avgiftsdirektoratet. p. 51.

45. Wikipedia. *White-box testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Structural_testing.

46. Wikipedia. *Black-box testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Black-box_testing.

47. Wikipedia. *Verification*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Verification.

48. Wikipedia. *Validation*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Verification_and_validation.

49. Wikipedia. *Verification and validation*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Verification_and_validation.

50. Wikipedia. *Systemutviklingsmetodar*. [cited 2010; Available from: http://no.wikipedia.org/wiki/Systemutvikling.

51. Wikipedia. *Testverktøyet JUnit for einheitstesting av Javakode*. [cited 2010; Available from: http://junit.sourceforge.net/.

52. Wikipedia. *Testverktøyet Selenium*. [cited 2010; Available from: http://selenium.seleniumhq.org.

53. Gleichmann, M. *Testverktøya FIT og jFIT*. [cited 2010; Available from: http://gleichmann.wordpress.com/category/fit/.

54. http://sourceforge.net/. *FindBugs - Find Bugs in Java Programs*. [cited 2010; Available from: http://findbugs.sourceforge.net/.

55. http://sourceforge.net/. *Testverktøyet PMD for Java Program*. [cited 2010; Available from: http://pmd.sourceforge.net/.

56. Wikipedia. *Integrated development environment (IDE)*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Integrated_development_environment.

57. Wikipedia. *FitNesse - eit automatisert testverktøy for testing av programvare*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Fitnesse.

58. Wikipedia. *RSpec - rammeverk for testing av programmeringsspråket Ruby*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/RSpec.

59. Wikipedia. *V-Model*. 2008 [cited; Available from: http://en.wikipedia.org/wiki/V-Model.

60. Wikipedia. *Acceptance testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Acceptance_testing.

61. Dataforeningen. *Software 2010*. 2010 [cited 2010; Available from: http://www.dataforeningen.no/software-2010.148681..html.

62. Grove_consultants. *Presentasjon av Lloyd Roden*. 2010 [cited 2010; Available from: http://www.grove.co.uk/lloyd.htm.

63. Roden, L., *Top Challenges (or opportunities) in Testing Today*, in *Software 2010*. 2010, Den norske dataforening: Oslo.

64. Roden, L., *Test Estimation - a pain or painless experience*, in *Software 2010*. 2010, Den norske dataforening: Oslo.

65. Wikipedia. *Scope creep - fenomenet at omfanget av kravspesifikasjonen aukar undervegs*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Scope_creep.

66. Wikipedia. *Regression testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Regression_testing.

67. Fowler, M. *The XP 2002 Conference*. 2002 [cited 15.02.2010]; Available from: www.martinfowler.com/articles/xp2002.html.

68. Grove_consultants. *Presentasjon av Julie Gardiner på nettsida til Grove Consultants*. [cited 2010; Available from: http://www.grove.co.uk/julie.htm.

69. Gardiner, J., *Risk Based Testing in any lifecycle*, in *Software 2010*. 2010, Den norske dataforening: Oslo.

70. Wikipedia. *Equivalence partitioning*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Equivalence_partitioning.

71. Wikipedia. *Decision tree*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Decision_tree.

72. Wikipedia. *All-pairs testing or pairwise testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/All-pairs_testing.

73. Wikipedia, *Ad hoc testing*. 2010.

74. Bakken, A., *Test og QA. Sammen er vi dynamitt!* 2010, Software 2010 - Den norske dataforening: Oslo.

75. Wikipedia. *Static testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Static_testing.

76. Eidsaunet, I. and Å. Ravik, *Effektiv testprosess under tids- og ressurspress*. 2010, Sogeti: Oslo.

77. Gjertsen, M. and B.H. Jørgensen, *Erfaringer med test i SPKs smidige prosjekt PERFORM*. 2010, Statens Pensjonskasse: Oslo.

78. Wikipedia. *Scrum - lettvektsmetode for iterativ og inkrementell systemutvikling*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Scrum_(development).

79. Wikipedia. *Code refactoring*. [cited 30.04.2010]; Available from: http://en.wikipedia.org/wiki/Code_refactoring.

80. Graham, K., *Hvordan finne ut hva du bør og kan effektivisere?* 2010, Software 2010 - Den norske dataforening: Oslo.

81. Sogeti. *TPI-modellen og testmetodikken TPI Next* [cited 2010; Available from: http://www.sogeti.no/no/Vare-tjenester/Tjenester-Norge/Software-Control-Testing/TPI/.

82. Dilorio, F.C., *The SAS Debugging Primer*, in *SUGI 26*. 2001: Long Beach, California.

83. Hetzel, W.C., *The complete guide to software testing*. 1988, Wellesley, Mass.: QED Information Sciences. ix, 284 s.

84. Pettichord, B., *Schools of Software Testing*. 2007.

85. Jorgensen, P., *Software testing: a craftsman's approach*. 2008, Boca Raton, Fla.: Taylor & Francis. 412. s.

86. Wikipedia. *Legacy code*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Legacy_code.

87. Bach, J. and P.J. Schroeder. *Pairwise Testing: A Best Practice That Isn't*. in *Twenty-second annual pacific northwest Software Quality Conference*. 2004. Oregon Convention Center Portland, Oregon.

88. Whitmill, K. *Becoming a Better Tester Staying Relevant*. in *Twenty-second annual pacific northwest Software Quality Conference*. 2004. Oregon Convention Center Portland, Oregon.

89. Bach, J. (2002) *Exploratory Testing Explained*. **Volume**, 10

90. Wikipedia. *Heuristic evaluation*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Heuristic_evaluation.

91. Zylberman, A., *Automated Exploratory Testing*. 2009.

92. Bach, J. *Exploratory Testing and the Planning Myth*. **Volume**,

93. Agruss, C. and B. Johnson (2000) *Ad Hoc Software Testing*. **Volume**,

94. Wikipedia. *Alpha testing and Beta testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Alpha_testing#Alpha_testing.

95. Bach, J., J. Bach, and M. Bolton (2009) *Exploratory Testing Dynamics*. **Volume**,

96. support.sas.com. *Global oversikt over brukargrupper for SAS*. 2010 [cited 2010; Available from: http://support.sas.com/usergroups/index.html.

97. www.sesug.org. *Presentasjon av SESUG (SouthEast SAS Users Group)*. 2010 [cited 2010; Available from: http://www.sesug.org/SESUGOrganization/index.htm.

98. www.nesug.org. *Presentasjon av NESUG (NorthEast SAS Users Group)*. 2010 [cited 2010; Available from: http://www.nesug.org/.

99. Wikipedia. *Maintainability*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Maintainability.

100. Wikipedia. *Dynamic testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Dynamic_testing.

101. support.sas.com. *SAS sin systemdokumentasjon av kva SAS-loggen er*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/lrcon/61722/HTML/default/a000998454.htm.

102. Ratcliffe, A. *Proactive Debugging*. in *SEUGI '99*. 1999.

103. Ratcliffe, A., *Debugging Made Easy*, in *SESUG 2001*. 2001, SESUG.

104. Flavin, J.M. and A.L. Carpenter. *Taking Control and Keeping It: Creating and Using Conditionally Executabel SAS Code*. in *SUGI25*. 2000. Indiana Convention Center, Indianapolis, Indiana: SAS Users Group International Conference.

105. Schlegelmilch, G.E. *Structuring Base SAS for Easy Maintenance*. in *SUGI 26*. 2001. Long Beach Convention Center, California: SUGI.

106. SAS_Institute, *Kurset "SAS Macro Language 2: Developing Macro Applications, 2-dagers kurs i desember 2009*. 2009, SAS Institute: Oslo.

107. Delwiche, L.D. and S.J. Slaughter, *Errors, Warnings and Notes (oh my). A practical Guide to Debugging SAS programs.*, in *SUGI 28*. 2003: Seattle, Washington.

108. Mitchell, R.M. (2006) *Finding Your Mistakes Before They Find You: A Quality Approach For SAS Programmers*. **Volume**, 8

109. Howard, N. and M. Gayari (2000) *Validation, SAS, and the Systems Development Life Cycle: An Oxymoron?* **Volume**,

110. Stojanovic, M. *SAS® Log Summarizer – Finding What's Most Important in the SAS® Log*. in *SESUG 2008*. 2008. TradeWinds Island Resorts, St. Pete Beach, Florida: SESUG Proceedings, distribuert av Institute for Advanced Analytics (http://analytics.ncsu.edu).

111. Smoak, C.G. *A Utility Program for Checking SAS Log Files*. in *SUGI 27*. 2002. Conference Chair, Cyndie Gareleck, Orlando, Florida: SAS User Group International (SUGI).

112. Fehd, R.J. *tip: macro LOGSAVE v1*. [www.listserv.uga.edu] 2001 [cited 2010 30.03.2010]; Available from: http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0110d&L=sas-l&F=&S=&P=8887.

113. Terjeson, M. *Log file reading program*. [www.listserv.uga.edu] 2004 [cited 30.03.2010]; Available from: http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0411E&L=sas-l&P=16645.

114. Yellanki, J.N. (2007) *Importance of Warnings and Notes messages from SAS log*. **Volume**,

115. Buck, D. and L. Stewart, *Explaining Unexpected Log Messages and Output Results from DATA Step Code*, in *SUGI 30*. 2005: Philadelphia, Pennsylvania.

116. Truong, S. *Making Code Review Painless*. in *WUSS 12 - Western Users of SAS Software*. 2004. Pasadena: Meta-Xceed, Inc, Fremont, CA.

117. Li, T. and J.K. Troxell. *A Macro to Report Problematic SAS Log Messages in a Production Environment*. in *North East SAS User Group*. 2001. Baltimore, Maryland: North East SAS User Group 2001.

118. Humphreys, S. *%LOGCHECK: a Convenient Tool for Checking Multiple Log Files*. in *The Pharmaceutical Industry SAS Users Group 2008*. 2008. Atlanta, Georgia: The Pharmaceutical Industry SAS Users Group 2008.

119. Foley, M.J. *Cutting the SAS LOG down to size*. in *South East SAS Users Group 2004*. 2004. Nashville, Tennessee: South East SAS Users Group 2004.

120. Sherman, P.D. *Intelligent SAS Log Manager*. in *SouthEast SAS Users Group 2007*. 2007. Hilton Head, South Carolina: SouthEast SAS Users Group 2007.

121. Svendsen, P., *Testing av SAS-applikasjon*, Riksarkivet, Editor. 1999: Oslo. p. 3.

122. Haugen, J.A., *Test av SAS-applikasjonen*, Riksarkivet, Editor. 1999: Oslo. p. 2.

123. Nygaard, L., *Testplan for SAS-verktøyet*, Riksarkivet, Editor. 1999: Oslo. p. 9.

124. Ulriksson, P., *Forslag til testsystem*, Riksarkivet, Editor. 2001: Oslo. p. 1.

125. Pettersen-Dahl, T., *Testplan for Arkade*, Riksarkivet, Editor. 2002: Oslo. p. 13.

126. Ulriksson, P., *Liste over feil/endringer i Arkade, sist oppdatert 12.12.2002*. 2002: Oslo.

127. Knapp, P. *Debugging 101*. in *SUGI 29*. 2004. Montreal, Canada.

128. Burlew, M., *Debugging SAS Programs: A Handbook of Tools and Techniques*. 2001: SAS Institute Inc. . 360.

129. Czerwonka, J. *Pairwise Testing*. 2009 Juli 2009 [cited; Available from: http://www.pairwise.org/.

130. Dustin, E., *Ortogonally Speaking*, in *STQE, the software testing and quality engineering magazine*. 2001.

131. Cohen, D.M., et al., *The Combinatorial Design Approach to Automatic Test Generation.* IEEE Softw., 1996. **September 1996**: p. 83-87.

132. Kuhn, R., et al., *Combinatorial Software Testing.* 2009.

133. Cohen, D.M., et al., *The AETG System: An Approach to Testing Based on Combinatorial Design.* IEEE Softw., 2007. **23**(7).

134. Satisfice_Inc. *Testverktøyet "Allpairs" for design av parvis testing*. [cited 2010; Available from: http://www.satisfice.com/tools.shtml.

135. Wikipedia. *GNU General Public License*. [cited 2010; Available from: http://en.wikipedia.org/wiki/GNU_General_Public_License.

136. Microsoft. *Omtale av programmet PICT som verktøy for generering av testtilfelle for parvis testing*. [cited 2010; Available from: http://msdn.microsoft.com/en-us/library/cc150619.aspx.

137. CaseMaker. *Programmet "Casemaker" for design av parvis testing*. [cited 2010; Available from: http://www.casemaker.eu/.

138. Wikipedia. *File viewer*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/File_viewer.

139. Ulrikson, P., *Systemdokumentasjon for Arkade, versjon 3.1.10*. 2005.

140. Arkivverket. *Arkadukt - omtale av programmet på Arkivverket si nettside*. 2010 [cited 2010; Available from: http://www.arkivverket.no/arkivverket/Arkivbevaring/Elektronisk-arkivmateriale/Testverktoey/Arkadukt.

141. Dilorio, F.C., *The elements of SAS programming style*, in *Proceedings of the Twenty First Annual SAS Users Group International Conference, SUGI 21*. 1996, SAS Inst. p. 385-9.

142. Riba, D.S. *How to Use the Data Step Debugger*. in *SAS Users Group International 2000*. Indiana Convention Center, Indianapolis, Indiana: SAS Users Group International.

143. Schreurs, N., *Grønt lys for skandaleprosjekt*, in *Computerworld*. 30.04.2010, Computerworld.

144. Wikipedia. *Pair programming*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Pair_programming.

145. Miller, J., *Applying meta-analytical procedures to software engineering experiments.* Journal of Systems and Software, 2000. **54**(1): p. 29-39.

146. Dataforeningen. *Faggruppa for Software Testing i Den Norske Dataforening*. [cited 2010; Available from: http://www.dataforeningen.no/software-testing.134251.no.html.

147. Wikipedia. *Test method*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Test_method.

148. Wikipedia. *Agile testing*. 2010 [cited 2010; Available from: http://en.wikipedia.org/wiki/Agile_testing.

149. Koomen, T. and M. Pol, *Test Process Improvement - A practical step-by-step guide to structured testing*. 1999: Addison-Wesley. 215.

150. Ulrikson, P., *Bruksdokumentasjon for Arkade*. 2003.

151. Wikipedia. *Document Type Definition (DTD)*. [cited 2010; Available from: http://no.wikipedia.org/wiki/Document_Type_Definition.

152. Wikipedia. *Wikipedia sin definisjon av termen "data set"*. [cited 2010; Available from: http://en.wikipedia.org/wiki/Data_set.

153. Interkommunalt_arkiv_for_Møre_og_Romsdal. *Metadata standarder referert på nettsida til IKA Møre og Romsdal*. [cited 2010; Available from: http://www.ikamr.no/content/view/full/449.

154. Hamilton, P. *%SYSFUNC: Extending the SAS Macro Language*. in *Pacific Northwest SAS Users Group 2005*. 2005. Tacoma (Sheraton): The Pacific Northwest SAS Users Group.

155. Burlew, M., *SAS Macro Programming Made Easy*. 2 ed. 2006, Cary, NC, USA: SAS press. 426 sider.

156. Winn, T.J. *Guidelines for Coding of SAS Programs*. in *SAS Users Group International 29*. 2004. Montreal, Canada: SAS Users Group International 29.

157. Carpenter, A., *Carpenter's Complete Guide to the SAS Macro Language*. 2004: SAS Press. 476.

158. SAS_Institute. *SAS OnlineDoc 9.1.3*. 2009 [cited 2009; Available from: http://support.sas.com/onlinedoc/913/docMainpage.jsp.

159. SAS_Institute. *General Macro Debugging Information*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a001302411.htm.

160. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MFILE*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a000209304.htm.

161. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MCOMPILENOTE*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a002475233.htm.

162. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MLOGIC*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a000543630.htm.

163. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MPRINT*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a000209298.htm.

164. Ulrikson, P., *Arkade. Prosesseringssystem for databaser, version 3.1.10*. 2005. p. 156.

165. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MLOGICNEST*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a002475221.htm.

166. SAS_Institute. *SAS sin systemdokumentasjon for systemopsjonen MPRINTNEST*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a002475227.htm.

167. SAS_Institute. *SAS sin systemdokumentasjon av systemopsjonen SYMBOLGEN*. 2010 [cited 2010; Available from: http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/a000543586.htm.

168. Simon_Fraser_University. *SAS Language Reference Dictionary: Functions and CALL Routines by Category*. 2010 [cited 2010; Available from: http://www.sfu.ca/sasdoc/sashtml/lgref/z0245860.htm.

Vedlegg

Sidan vedlegga tilsaman er på 175 sider, er det valt å trekka dei ut frå den trykte utgåva av masterrapporten. Dei ligg vedlagt som ei pdf-fil på ei CD-plate. Sidenummereringa for den elektroniske fila med vedlegg startar på side 140, som er same sidetal som vedlegga ville starta på om dei hadde komme etter rapporten.