1 Bakgrunn

1.1 Hvorfor lage AuRa?

Det finnes i dag mange rapporteringsoppgaver som må gjøres på et jevnlig basis [whitepapers og slikt kan siteres, er det en ok ting å gjøre?], og disse tar tid og krefter, og verre er, går ut på å hente ut data fra datamaskinen, lime det sammen i et format andre kan lese og sende det av gårde. Dette tar tid som kunne produktivt brukes på andre ting. Videre er eksisterende løsninger deler av store automatiseringssystemer som er ment til å integrere mot andre løsninger, men ikke å bli integrert mot utenfra. De passer dermed ikke til å automatisk generere dokumenter som en del av en annen prosess, med mindre de selv styrer denne prosessen. Dette låser en ned til spesifikke løsninger, og gjør en mindre fleksibel. En annen løsning har tidligere vært å skripte vha. Python, PERL, Bash eller lignende. Dette er mer fleksibelt, og lar deg kompartmentalisere rapportgenereringen inn i logiske enheter relativt enkelt, men introduserer andre problemer i stedenfor: De har ofte dårlige teknikker for innpakking av kode, de blir fort komplekse, og de må vedlikeholdes på lik linje med all annen kode. De er dermed heller ikke perfekte. En kan selvsagt skrive sitt eget språk for å håndtere denne type data, men da får en et nytt problem. En må nå vedlikeholde et helt språk, og med mindre en får andre til å bruke det, får en problemet med å lære nye mennesker opp i språk.

Oppgaven vil prøve å takle problemet en bit på vei med å introdusere AuRa. Som er både et språk, men også et rammeverk for å produsere rapporter. AuRa er ment til å håndtere problemene som skripting medfører ved hjelp av følgende designvalg:

- Språket er med vilje holdt enkelt. Det er ingen kompliserte ting i språket som helhet. Du snakker om tekst og*eller filer, og du får ut en eller flere filer.*
- Språket velger alltid klarhet framfor å være kortfattet. Du skriver gjerne kode en gang, men du leser den fire. Å gjøre det lettere for de som skal vedlikeholde programmene beskrevet i AuRa er en stor fordel.
- Språket er valgt til å enkelt kunne genereres programmatisk. På den måten kan en skrive verktøy som bruker AuRa uten å være bundet til AuRa, eller måtte være en del av AuRa. På den måten kan en ha et samfunn rundt denne løsningen.
- Språket vil beskrive hver bit av programmet som en fil. Denne filen må dermed enten finnes eller genereres på forhånd. Dette kan gjøres

ved hjelp av til dømes Make, SCons eller Rake som kan enten kalle programmer som genererer filen, eller kjøre kommandoen som genererer den selv. En framtidig utgave av AuRa burde også støtte denne typen enkel oppførsel, men for å bli ferdig med prosjektet støttes ikke dette i den nåværende versjonen.

- Rammeverket vil også legge opp til å bruke andre løsninger som allerede finnes, er velprøvde og fungerende der det lar seg gjøre. Det er til dømes ingen vits i å lage et system for å kjøre en AuRa-fil ved en viss tid eller mellom et visst intervall når cron(8) allerede finnes.

- Rammeverket legger opp til at hver genererte bit av innhold blir separert fra andre biter av innhold. Dette har konsekvenser, både positivt og negativt, men det fremmer igjen synspunktet at det er viktigere å være enkel å vedlikeholde enn å være rask å skrive.

  Et annen type problem er utdataformater. De er ofte relativt prisgitt det som støttes av plattformene. Dersom en bruker et automatisk system er du låst til de formatene som systemet støtter. Skripter du er du prisgitt det som er tilgjengelig. Det kan være alt fra HTML-templates til LaTeX hånd-hacking. Ingen av delene er spesielt fantastiske. På den ene siden kan man generere et format og så bruke et verktøy som Pandoc til å konvertere, men det blir fort problematisk om du har krav som Pandoc ikke kan eller vil støtte.

  For å unngå denne problemstillingen er AuRa delt opp i to deler ala Gnu Compiler Collection, og fungerer som følgende: Først blir rapporten generert som normalt, og kompilert til et mellomformat. Dette mellomformatet er uttrykt i standard symbolske uttrykk fra Lisp, som gjør det lett å lese, parse og skrive. Dette mellomformatet er instruksjoner om hvordan rapporten skal se ut, og inneholder alle data som behøvs for å generere det endelige dokumentet. Dette dokumentet blir så gitt videre til en ny kompilator. Denne kompilatoren genererer den endelige rapporten, og forskjellige generatorer vil generere forskjellige typer og formater av rapporter. En kan tenke seg klartekstrapporter, e-post rapporter som formaterer om til en e-post som kan sendes direkte til mottakere, ODF-rapporter |ODF = Open Document Format| for å overholde EU-standarden for offentlig kommunikasjon, eller LaTeX rapporter, som kan bli kompilert med LaTeX kompilatorer om til PDF eller PostScript. Mulighetene er mange, men det beste med denne typen løsning er at om et nytt format skulle ta verden med storm, kan en enkelt ta hånd om dette formatet vha. å skrive en ny

viderekompilator som en plugin for AuRa.

En tredje stor fordel med AuRa er at alle kildene som oppgis er uavhengig av hverandre. Dermed kan en teoretisk generere delene av rapporten paralellt, og kan dermed gjøre ting enda raskere og mer effektivt enn en vanlig seriell kompilering. Der en i en tradisjonell skriptingløsning ville kunne kjørt GNU Make, og latt den gjøre alle oppgavene serielt, kan en nå paralellisere denne type oppgaver i AuRa. Dette er ikke støttet per i dag, men er en opplagt forbedring som kan gjøres i nyere utgaver.

Den fjerde store fordelen som allerede er nevnt er at det er lett å skrive verktøy til denne type språk. Det er altså ikke slik at en må skrive AuRa for hånd, men kan bruke eventuelle GUI-verktøy til å sette opp prosjektet, si hva som skal gjøres, og så generere AuRa-filene fra disse dataene. Tekstfilene som blir genererte vil fortsatt være forståelige, siden formatet er designet til å være enkelt. Store rapporter med hundrevis av deler kan være kjedelig å lese gjennom, men er likeledes ikke vanskelig å skumme eller søke gjennom.

Sett under ett gir dette en fleksibilitet som per dags dato mangler i andre større løsninger. Satt sammen med et enklere språk å skrive i, som er lettere å vedlikeholde og kan bli generert av brukervennlige programmer minskes problemene med tradisjonelle skriptløsninger. Samtidig har vi en økt smidighet sammenlignet med mekanismene som tilbys av automatiseringsløsninger.

1.2 Hva finnes fra før?

1.2.1 Automate (Kilde: http:

www.networkautomation.com)

Automate er et system for å generere rapporter automatisk vha. en GUI, hvor en kan dra og slippe rapportenheter. Systemet lover automatisk generering av rapporter uten å måtte skrive en linje kode, for alle støttede filkilder. Rapportgenereringen er en integrert del av en større pakke de tilbyr.

De viktigste forskjellene mellom AuRa som er foreslått, og AutoMate sin rapportgenerering er:

- AutoMate er et ferdig program fra begynnelse til slutt, mens AuRa er ment til å integreres med andre løsninger.

- AutoMate tilbyr ODS formatet på dokumentene, mens AuRa tilbyr alle formater som kan støttes med en kompilator-plugin.
- AutoMate er ikke ment til å kunne programmeres, mens for AuRa er det et poeng at du kan programmere det enkelt og smertefritt. Til dømes, i deres case study påpekes det at AutoMate kan lett stilles inn til å trekke ned data, lage en rapport, og skrive den ut. I AuRa, ville en tilsvarende løsning vært å bruke lpr/lpd og cron til å automatisere mtp tid og utskrift.

For å oppsummere, er AuRa mer som et forenklet skriptspråk, med enkle kommandoer ment for å gjøre skriving av en spesiell type skript enklere og raskere. Der AutoMate er en programpakke som gjør jobben fra ende til ende, er AuRa ment til å være et verktøy som integreres med andre verktøy til en komplett pakke.

Det finnes også andre kommersielle pakker for generell automasjon, som til dømes AutomationAnywhere, men de er tilstrekkelig like til at de ikke trenger egen behandling.

1.2.2 Pandoc

Pandoc er et program som oversetter fra et dokumentformat til et annet. Derfor er det interessant fordi det parser dokumenter og lager dokumenter ut av dem. Per i dag kan du selvsagt skripte handlinger som å hente data ut fra en server, du kan skripte numerisk analyse vha. statistiskkprogrammer som til dømes R, og du kan lime sammen mange filer til en stor fil og sende det til et program, som for eksempel en LaTeX kompilator, som kan lage pdf-en din.

Men du må altså gjøre det for hånd. Og selv om du kan sette opp make til å kjøre skriptene som henter ut og behandle dataene, og deretter lime det sammen via cat til en enhetlig fil som kan kompileres, er du begrenset til et format som du kan evt. kompilere vha. programmer som for eksempel Pandoc.

En kan også tenkes å bruke AutoMate som tidligere nevnt, og konvertere OpenDocument formatet den gir deg til PDF, eller legge til forståelse for mellomformatet til AuRa i Pandoc, og så bruke Pandoc til å konvertere. Pandoc er et nyttig program, som er interessant for dets evne til å forstå dokumenter, men det er altså ikke et system for å generere dokumenter, men å knovertere dem.

1.3 Inspirasjoner

AutoMate og dets konkurrenter har tilbudt løsninger som er store og integrerte. De fungerer som monolittiske entiteter, og er dermed ikke enkle å integrere i andre løsninger. De er ment til å integrere andre løsninger til seg, vha. støtte for å bevege musen, trykke på tastaturet, med mer.

Til sammenligning har en andre systemer som Ant og Make som er bygd opp rundt UNIX-filosofien [**?**], der programmer kan integreres med og mot hverandre enkelt og ukomplisert.

1.4 Hva er nytt?

Det finnes rapportgenereringsverktøy til salgs per dags dato. Det som er nytt med dette verktøyet er at det er utformet som et enkelt språk. Dette språket er lett å parse, det er lett å skrive, og det er lett å lese. Denne enkelheten gjør at andre verktøy kan bruke systemet til å produsere rapporter med. En kan også skrive sine egne "programmer" for hånd på den måten. For eksempel kan en skrive en masteroppgave i Markdown, og kompilere med AuRa, og få ut en PDF. Denne enkle integrasjonen med og mot andre verktøy gir nye muligheter for integrering og oppgaveflyt. Mens andre systemer lar en kale andre programmer for å lage rapporten, kan en her generere rapporten programmatisk.

1.5 Hva er ikke nytt?

Ideen om å ha et enkelt språk som beskriver handlinger er selvsagt ikke nytt, og skriptspråk som for eksempel BASH eller PERL er eksempler på slikt.

GNU Make og Ant er begge byggesystemer som kunne blitt brukt i en del av systemet selv. Disse systemene har metoder for å definere mål som du deretter kan be dem utføre. Til dømes kan du be Make om å bygge et prosjekt, installere det, avinstallere det, rydde opp midlertidige filer og annet.

I rapportgeneratoren derimot, peker du på en fil som definerer prosjektet, og sier hvilket format som skal benyttes. Det er bare ett mål. Make er veldig fint til å kjøre kommandoer før rapporten bygges, da kan en sikre seg ferske data, slik at en alltid bruker oppdaterte tall. Dette kunne blitt bygget inn i systemet, men tidsbegrensinger og tilstedeværelsen av make gjorde det unødvendig.

Plugin-systemet er mye likt Emacs sitt system, som er enkelt å drifte, selv om det ikke er ferdig artikulert. Det har manuell håndtering av avhengigheter,

og hver plugin antas å kun avhenge av hovedsystemet, eller komme med avhengighetene selv.

1.6 Hva er målet med oppgaven?

Målet med oppgavene er altså å argumentere for et nytt språk. Et språk som kan brukes til å generere rapporter automatisk og er utelukkende fokusert på dette formålet. Dette språket lar en generere rapporter enkelt og løser dermed problemet med generiske shellscripts ved å være lettere å vedlikeholde, ved å være et enklere språk med tydeligere syntaks. Videre vil et enkelt språk la en generere rapportbeskrivelser programmatisk vha. tredjeparts programmer som kan integrere mot dette systemet.

Rapporter her er ikke bare ting som salgsrapporter eller ledige sykesenger per dag eller slikt, men kan også være resultater av undersøkelser med mer. Ved å velge å ikke tilby mer enn nødvendig i språket, men heller tilby å bruke andre språk er dette også et kall til UNIX-tradisjonene med flere små programmer som er lette å lære, bruke, og koble sammen.

Hvis du allerede bruker R, hvorfor ikke fortsette å bruke R, og heller bruke resultatene i en rapport uavhengig? Og hvorfor ikke ha et byggesystem for slike rapporter? Dersom du kan ha kontinuerlig integrering (Continous Integration) i alle ledd helt ned til rapporten som leveres til slutt, vil det være bedre for alle involverte. Du må selvsagt ikke bruke R. Du kan også bruke Python, Julia, Java, eller andre programmer. Poenget er å appellere til valgfrihet, og åpne opp for bruken av flere verktøy, istedenfor å låse brukere fast til bare et av dem.

Videre er det også et argument, ikke bare for et språk, men også et rammeverk for å skrive rapporter i. I stedenfor å beskrive rapporter for hånd kjedsommelig, vil jeg argumentere for at å ha et system som gjør det lett å automatisere denne type oppgaver vil være til det felles beste.

Det er dermed ikke et argument for at AuRa og dets konvensjoner er de beste konvensjonene som kan tenkes, men et argument for at å ha konvensjoner og et språk er et steg opp fra å ikke ha dem. Helintegrerte programpakker som AutoMate kan bruke systemer som AuRa, men det kan også enklere systemer som Makefiler, eller shellscript eller et fullblods skrivebordsprogram som trenger å lage dokumenter.

Byggesystemer som Make, Ant, Rake og lignende har et bruksområde for å generere informasjon, tilbakemeldinger til programmerere og lignende. AuRa vil kunne gjøre dette arbeidet enklere ved å kunne generere e-poster, html-sider eller PDF-rapporter som kan sendes til relevante mennesker. Generiske rapporteringsoppgaver som for eksempel ledige senger ved sykehus, en

pasientjournal for de siste 14 dagene pasienten har blitt innlagt, med journal og notater fra sykesengen vil kunne lages automatisk og sendes til legen uten at hun må etterspørre dette.

Det er mange muligheter med et slikt system, og å gjøre det enklere å få informasjon ut fra datamaskiner og inn i hendene til mennesker som trenger det vil kunne gjøre livet enklere for mange.

- Eller evt. et kjærlighetsbrev til denne tradisjonen, men dette er en seriøs akademisk tekst, blottet for humor og selvinnsikt.

1.7 Materialer

Systemet er for det meste skrevet i Common Lisp, med hjelp fra verktøy og biblioteker fra Quicklisp. Alle programmene er åpen kildekode, lisensiert med frie lisenser som BSD eller GPL og tilgjengelige vederlagsfritt.

1.7.1 Steel Banks Common Lisp

Implementasjonen av Common Lisp er SBCL, som er basert på Carnegie Mellon University Common Lisp (CMUCL) og deler bugfixer med hverandre. Hovedforskjeller inkluderer støtte for native-tråding på Linux, som gir en økt mulighet for paralellisering av arbeidsmengder sammenlignet med CMUCL. SBCL er ansett som en robust og rask implementasjon av Common Lisp, og er valget av Lispimplementasjon på Alioths "Computer Language Benchmarks Game", der den er et av de raskere språkene.

Common Lisp er et multiparadigmespråk, som tilbyr objektorientering, funksjonell programmering med høyere ordens funksjoner, og andre metodologier som ønskes. Språket tilbyr også makroer som gjør det mulig å legge til eller endre syntaxen for språket, eller gi språket nye operasjoner det ikke kunne før.

Utgaven av Steel Banks Common Lisp som er brukt i systemet er SBCL 1.2.3.debian (for AMD64).

1.7.2 Quicklisp

www.quicklisp.org

Quicklisp er et system for å håndtere avhengigheter i Common Lisp. Det kan sammenlignes med Apache Ivy eller Apache Maven, med noen forskjeller.

- Apache Maven er ment til å håndtere hele prosesser, mens Quicklisp er begrenset til å håndtere avhengigheter.

- Apache Maven og Apache Ivy kjører ved bygging, mens Quicklisp er tilgjengelig mens programmet kjører.
- Java og Common Lisp bygges på forskjellige måter, så mens Apache Maven har sitt eget byggesystem, og Apache Ivy er tett integrert med Apache Ant, er Quicklisp integrert mot ASDF, som er et byggeverktøy for Common Lisp.

Siden Quicklisp er tilgjengelig ved kjøretid kan en bruke verktøyet fra Read Eval Print Loop (REPL) promptet, og dermed bruke Quicklisp som et verktøy for utforskende programmering. For å støtte denne bruken har quicklisp også funksjoner for å bla gjennom og søke opp i pakkebrønnene (Eng: repositories) sine etter pakker med spesifikke navn.

Selv om prosjektet enda er i beta, håndterer det allerede over 1300 biblioteker i pakkebrønnen sin. Det har også integrasjon mot SLIME, en vanlig Common Lisp IDE laget for GNU Emacs, skrevet i Emacs Lisp.

Fordi det er tilgjengelig ved oppstart kan en også be Quicklisp om å oppdatere alle evt. avhengigheter programmatisk, for eksempel ved oppstart, og å laste dem ned om de ikke er funnet i systemet.

Dette gjør bygging av lisp-prosjekter med eksterne avhengigheter enklere enn å laste ned tarballer og installere dem ved hjelp av ASDF. Merk at Quicklisp integrerer seg selv mot ASDF, og bruker dette systemet til å registrere og bruke eksterne biblioteker.

Da Quicklisp oppdaterer seg selv, er nyeste utgave gitt ut siden 28. mai testet.

1.7.3 CL-PPCRE - Portable Perl-compatible revular expressions for Common Lisp

CL-PPCRE er en effektiv motor for tolking av perl-kompatible regulære uttrykk. Den benytter seg av Lispkompilatoren for effektivitet ved å kompilere mønstrene ned til maskinkode. Dermed istedenfor å bygge en regulær-tilstandsmaskin som en VM, vil den generere en tilstandsmaskin i maskinkode. Dermed vil den kunne kjøre raskere enn tilsvarende PERL-regexer. (Dersom det er interessant er Aho-Corasick algoritmen bak blant annet fgrep, og et sted å begynne.)

CL-PPCRE er brukt til å parse det regulære språket Markdown i kompilatoren.

Versjonen brukt er den nyeste per 28. mai.

1.7.4 Split-Sequence

Split-Sequence er et enkelt bibliotek for å dele opp sekvenser i delsekvenser. Brukes i parsingen av inndata i prosjektet.

Versjonen brukt er den nyeste per 28. mai.

1.7.5 GNU Emacs

GNU Emacs er et skriveprogram gitt ut av GNU, med ekstensive utvidelsesmuligheter. Den har muligheter for å skrive tekst i mange formater, blant annet LaTeX, Bibtex, Markdown, Common Lisp, med mer. Den har også blitt utvidet til å ha integrerte utviklingsmiljøer, heriblant SLIME. GNU Emacs 24.1 ble brukt til å skrive oppgaven og til å programmere med (ved hjelp av SLIME).

Versjonen av GNU Emacs brukt er GNU Emacs 24.3.1

1.7.6 SLIME

SLIME (Superior Interaction Mode for Emacs) er et integrert utviklingsmiljø for Common Lisp for Emacs. Den støtter både GNU Emacs og XEmacs. Den støtter også flere implementasjoner av Common Lisp, deriblant Steel Banks Common Lisp.

SLIME kan integreres med Quicklisp og Quickproject, selv om sistnevnte ikke ble brukt i dette prosjektet. Som integrert utviklingsverktøy hjelper det til med det meste en skulle ønske et integrert utviklingsmiljø gjorde, inkludert REPL-spesifikke ting som snarveier, setting av nåværende navnerom, med mer.

1.7.7 Ubuntu 14.10

Programmet ble utviklet til å kjøre på GNU/Linux maskiner på AMD64 baserte prosessorer. I begynnelsen på Ubuntu 14.04LTS (Trusty Tahr), men senere Ubzuntu 14.10 (Utopic Unicorn). Da Steel Banks Common Lisp er et abstraksjonsnivå på toppen av operativsystemet, og ingen kall til systemet gikk utover standard POSIX-kall burde det ikke være noen problemer med å kjøre på andre systemer, men dette er ikke testet, og integrasjonstesting mot andre systemer ligger utenfor det som er tid til å gjøre på dette prosjektet.

1.7.8 Git DVCS

Til håndtering av kildekode er Git brukt. Git er et distribuert system for versjonshåndtering av kildekode, og blir brukt her til å holde orden på oppgave

og kildekode.

Utgaven av Git som er brukt er git version 2.1.0

1.7.9 Data brukt til testing av systemet

Det er to hovedkilder til testdata til systemet. I tillegg til enhetstester som kjøres ved oppstart av systemet (da Common Lisp ikke har separerte tidspunkt for kjøring og kompilering som til dømes C, Java eller Ada har), brukes masteroppgaven her som rådata til systemet. Dette hjelper på motivasjonen til å få alle deler av systemet til å kjøre. I tillegg til dette har jeg fått låne data fra Peter Ellison til å generere rapporter fra, som jeg er svært takknemlig for. [TODO: Husk å putte ham på listen over folk jeg skal takke]

1.8 Metoder

1.8.1 Rational Unified Process

Til å utvikle systemet ble Rational Unified Process som beskrevet hos [**?**] valgt, med modifikasjoner innenfor det som er beskrevet som lovlig. For eksempel har prosjektet blitt delt opp i delprosjekter for å ha en bedre oversikt over hva som skal gjøres og når, og har begrenset bruken av diagrammer til å kommunisere intensjoner med, med den begrunnelse at det er et enmannsprosjekt. Rational Unified Process som beskrevet av [**?**]s muligheter for å modifisere utviklingsmetodologien mellom iterasjoner for å tilpasse prosjektets behov er blitt brukt flittig. Dette har gitt en bedre forståelse av forskjellige formelle behov. For eksempel har bruken av interaksjonsdiagrammer blitt sterkt begrenset, noe som gjorde at formgivingen av interaksjonen med systemet fikk en lavere prioritet enn den kanskje hadde trengt. På den annen side gjorde det økte fokuset på tekniske detaljer og integrasjonen av systemene at disse detaljene kom på plass på en effektiv og gjennomtenkt måte.

Skulle prosjektet blitt gjennomført på nytt igjen fra bar bakke, ville RUP blitt brukt igjen. Det var en overraskende smidig metodologi, med mange anbefalinger, og få absolutte regler. Metametodologien er en stor del av dette. Kravet om at metodologiske verktøy og virkemidler blir evaluert mellom iterasjoner med mulighet for å forkaste eller legge til slike verktøy er viktig. Den andre delen er det nøkterne iterative synet Rational Unified Process har på utvikling.

1.8.2 Objektorientert programmering

Objektorientering er en måte å innkapsulere data og funksjoner i objekter. Disse objektene representerer entiteter i systemdesignet ditt, og kan utføre handlinger (metoder) basert på hvilke funksjoner de har tilgjengelige.

I de fleste språk er disse metodene meldinger som sendes til objektene. (Til dømes kan en streng i Java bli bedt om å gjøre alle tegn til store bokstaver slik: "Java Streng".toUpperCase(), dette anses da som å sende en melding til strengen) Common Lisps objektsystem CLOS (Common Lisp Object System) er fundamentalt annerledes, da den baserer seg på et system med generiske funksjoner istedenfor å sende meldinger.

En vil dermed i CLOS definere klasser uavhengig av funksjonene som opererer på dem. For å benytte seg av polymorfisme definerer man en generisk klasse, som har en signatur, og en kan dermed *spesialisere* denne generiske klassen med en metode. Denne metoden vil da ta en spesifikk typesignatur til minst et av argumentene sine. Det er viktig å påpeke her at mens i andre objektorienterte språk som Java og C# vil en metode kun tilhøre en klasse (siden den mentale modellen er å sende en melding til en klasse), vil en metode i CLOS kunne tilhøre flere klasser samtidig. Derfor kan en definere metoder uavhengig av klasser.

Dette ser ikke særlig annerledes ut, en kan til dømes ha en klasse "Kjøretøy" med to underklasser "Motorsykkel" og "Lastebil". En kan deklarere en generisk metode ved hjelp av defgeneric slik:

```
(defgeneric start-motor kjoretoy
  (:documentation "Starter en motor til et kjoretoy. Vil kaste en error om
```

Og en kan videre spesialisere en metode for Lastebil således:

```
(defmethod start-motor ((kjoretoy lastebil))
    (lag-lyd "vrom vrom!"))
```

På dette punktet kan generiske funksjoner minne om Interface i Java og C#, men for funksjoner istedenfor. Men en kan også tenke seg følgende generisk funksjon:

```
(defgeneric kollider traffikant1 traffikant2
      (:documentation "Kjorer over en myk traffikant med et kjoretoy"))
```

Og da kan vi spesialisere på for eksempel en lastebil og en fotgjenger slik (antagelsen er at kjøretøy arver fra traffikant):

```
(defmethod kollider ((traffikant1 lastebil) (traffikant2 fotgjenger))
     (if (skadet traffikant2))
(tilkall-ambulanse))
```

På dette tidspunktet tilhører metoden klassene *lastebil* og *fotgjenger*. De er dermed spesialiserte på begge to, og hører ikke konseptuelt hjemme hos noen av klassene. Det kan virke som om dette er lastebilen som kolliderer med fotgjengeren, men hva om det er to lastebiler som kolliderer? Eller to fotgjengere? I språk som Java og C# får en da et dilemma om hva som skal skje, og hvor koden skal ligge. (En mulig løsning er Visitor-mønsteret I CLOS er dette bygget inn i systemet og en unngår dilemmaet.

[Jeg er ikke helt sikker på hvor mye jeg bør ha her. På den ene siden så er CLOS veldig forskjellig fra andre systemer, og CLOS bryter veldig hardt med den vanlige måten å gjøre OOP på. Det er ikke gitt at folk har en akademisk bakgrunn i hvordan OOP fungerer. På den annen side kan det fort bli litt mye.]

1.8.3 Design av systemet

Systemet er ment til å være enkelt å bruke. Men hvem er da brukeren tenkt å være? Svaret er at brukere er ment til å være andre som integrerer systemet inn i sine egne verktøy og bygger videre på dem. Derfor bør språket:

- Være enkelt å skrive maskinelt.
- Være lett å verifisere.
- Om det skal kunne utvides, må det kunne utvides via godt dokumenterte grensesnitt.
- Validering må kunne skje med gode tilbakemeldinger ved feil, slik at brukere kan varsles på en god måte.

Siden alt er tekst og språket er lingvistisk enkelt å beskrive, er det ikke vanskelig for verktøy å integrere mot systemet. Hver linje beskriver en og bare en del av rapporten helt uavhengig av andre deler. Dette gjør at en kan tenke seg enkle klasser ala:

```
public Interface RepGenEntity \{
  public String repGenString();
\}
```

```
public class MarkdownReport extends RepGenEntity \{
  public file markdownFile;
  @Override
  public String repGenString() \{
    return String.format("(markdown fil=\textbackslash "\%s\textbackslash "
  \}
\}
```

Og med dette kan du bruke Java sine innebygde GUI elementer til å generere hele greien. JFileChooser, JavaFX, hele pakken. Med denne type objekter enkelt og greit.

Du kan lage noe lignende et slikt system vha. standard UNIX-verktøy som Make og Bash. Disse programmene er standard vare, og har vært i bruk lenge og er relativt feilfri. Men det er likevel en forferdelig ide, fordi den jevne kontormedarbeider ikke sitter med en UNIX-maskin, og om de gjorde det (Mac, til dømes er da rimelig populært per i dag), så ville de ikke forventes å kunne bruke Make eller BASH. I tillegg ville en slik løsning være porøs og vanskelig å sette opp for forskjellige systemer eller behov. Det er lite til ingen inkapsulering, og det er heller ikke enkelt å integrere mot andre verktøy som det burde vært. Uttrykk strekker seg over flere linjer, en har semantisk betydning i indenteringen, og andre anakronismer. (Lenke mot manualer her?) Et eget subspråk for disse typer uttrykk hadde vært bedre. Da blir det enklere å generere uttrykk programmatisk, og det blir lettere for andre mennesker å lese og forstå det som står i uttrykkene, om ikke nødvendigvis skrive dem. (Domain spesific languages boken?)

Da blir spørsmålet hvilke datakilder som skal støttes, og på hvilket nivå dette skal skje. En kan tenke seg den enkleste muligheten kan være rik tekst i form av et Markdown-format, bilder i form av Portable Network Graphics (PNG) og tabeller i form av Comma Separated Values. Disse tre dekker et vidt spekter av datatyper. Spørsmålet blir så hvor en skal hente data fra. Den enkleste løsningen er å peke til eksterne småprogrammer som kan skrives ad-hoc. For eksempel kan en ha en to-tre markdownfiler et eksternt bilde, og en tabell. Tabellen blir generert av et script som snakker med en database, og bildet blir lagget av et statistikkprogram (R, SPSS eller andre) som skriver ut resultatene sine som en graf som lagres som et bilde. Med R kan dette gjøres via et standard script.

En kan da tenke seg følgende fil:

```
;; Erklaer type
(utdata format="LaTeX\textit{PDF")



;; Hent data og marker for sletting
(kjor fil="\textasciitilde }rapporter\textit{testrapport}statistikk.r")
(slett-etter-kjoring fil="\textasciitilde \textit{rapporter}testrapport\tex



(kjor fil="\textasciitilde }rapporter\textit{testrapport}hent-database-info
(slett-etter-kjoring fil="\textasciitilde \textit{rapporter}testrapport\tex



;; Inkluder materiale
(le(markdown fil="\textasciitilde }rapporter\textit{testrapport}innledning.
(bilde fil="\textasciitilde \textit{rapporter}testrapport\textit{graf-fig-
(markdown fil="\textasciitilde }rapporter\textit{testrapport}brodtekst.md")
(tabell fil="\textasciitilde \textit{rapporter}testrapport\textit{database-
(markdown fil="\textasciitilde }rapporter\textit{testrapport}konklusjon.md"
```

Dersom en setter alle kilder og kall til dem som uavhengige uttrykk, og lar alle slike uttrykk være så enkle som mulig, så blir dette en beskrivelse av hva som skal inn i rapporten. Rapporten kan så genereres til et mellomformat som ikke ser så aller verst ut. Men hva med sluttresultatet? Skal den lages som HTML, ren tekst, PDF via LaTeX DocBook, eller noe annet?

Dette kan da implementeres ved hjelp av back-ends til kompilatoren. Disse kan da lages etter behov, plugges inn og registreres, slik at de blir tilgjengelige for systemet. Dermed kan en utvide systemet til å møte nye utfordringer.

Det er selvsagt noen negative sider ved et slikt design som er holdt så enkelt som mulig. Dersom en skal sette inn et bilde må en klippe en tekstfil i to. Dersom en skriver for hånd kan dette bli noe irriterende i lengden. Men dersom en bruker verktøy til å generere med, kan en unnslippe problemet ved å verktøyet gjøre dette bak kulissene. Da vil en unngå en del av problematikken.

Dette designet ble ikke ferdig implementert, selv om dette var tanken bak. Det ville vært relativt enkelt å lage et verktøy som kunne satt opp den ovennevnte filen. Tilgjengelige utformater kunne blitt oppdaget ved å sende spørringer til systemet. Programmer som skal kjøres kunne blitt satt inn relativt enkelt gitt at brukerene visste hvilke filer en var interessert i. Det eneste som hadde vært igjen ville være å la brukere skrive tekst som skulle brukes, velge hvor filer skulle settes inn, og la dem redigere til de var fornøyd.

Flere mulige funksjoner ble overveid men til slutt forkastet:

- Automatisk generering, til dømes hver dag, eller annenhver time ble forkastet til fordel for å bruke innebygde systemer for akkurat dette.
- Flere datatyper som JPG, RTF eller lignende ble forkastet for å kunne fokusere på muligheter, og spare tid. Støtte kan evt. utvides senere.
- Flere utdatatyper var en mulighet, men ville tatt for lang tid å fullføre.

[Blant flere muligheter. Dette er en trist liste.]