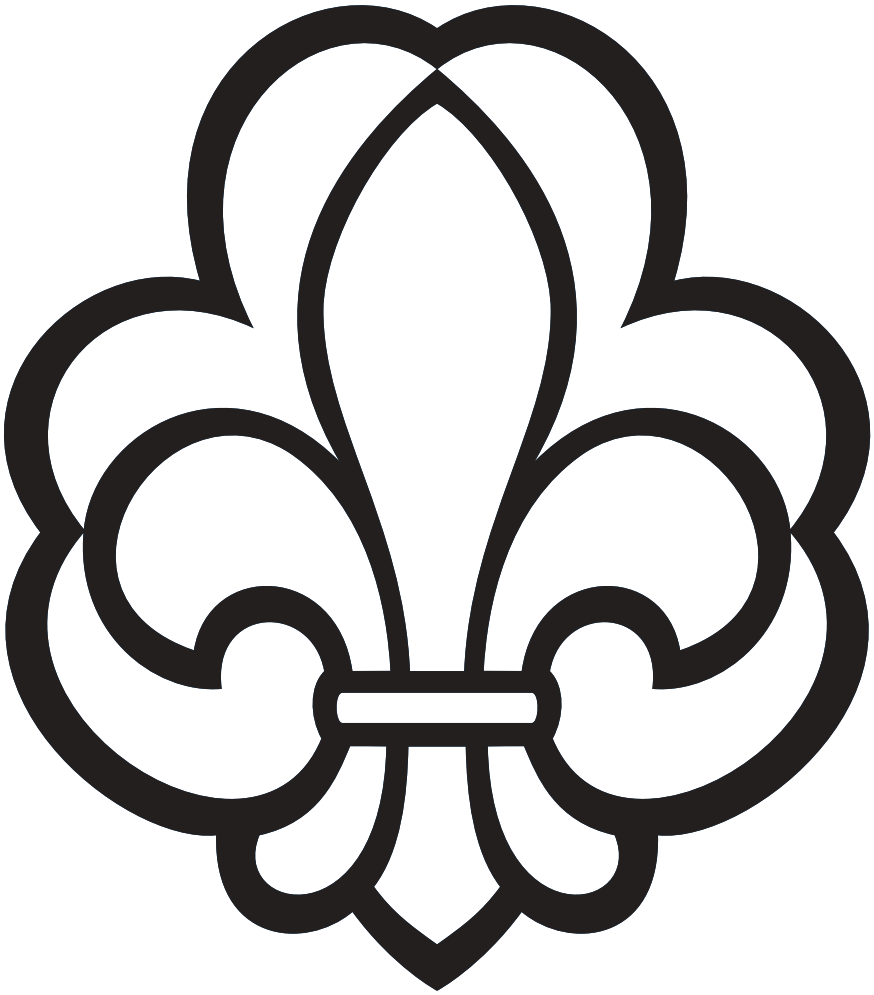
Digitalt Fotoarkiv

tok@itu.dk — **Troels Krogh**
mads@danquah.dk — **Mads Danquah**

Vejleder:

panic@itu.dk — **Arne John Glenstrup**

27. maj 2004



IT-Universitet i København
Internet- og softwareteknologi

Abstract

Rapporten beskriver designovervejelser til et fotoarkiv til Det Danske Spejderkorps, samt et design og implementering af et API til at tilgå datamodel sikkert. I forbindelse med designovervejelserne er der udført en afprøvning af ydelsen ved lagring af billeder i henholdsvis en database og et filsystem. Afprøvningen viste at et filsystem klart er at fortrække ved sekventiel adgang til filer. Der er lavet en delvis implementation af fotoarkivet, samt en afprøvning af APIet. I store træk er test en succes, men der er dog fejl.

Indhold

1 Forord

Denne rapport er udarbejdet i maj 2004 i forbindelse med et 4 ugers projekt på IT-Universitetet. Rapporten er uarbejdet af Mads Danquah og Troels Krogh, der går på henholdsvis 1 og 2 semester på IT-Universitetets Internet- og softwareteknologi linie. Deltagerne i projektet vil gerne benytte lejligheden til at sig tak til vejleder Arne John Glenstrup for god og kyndig vejledning. Endvidere skal Det Danske Spejderkorps have tak for lån af lokaler i projekt perioden.

I den oprindelige projektaftale indgik der at der skulle implementeres en fuldt fungerende prototype. Vi har pga. tidsmangel ikke kunne implementere hele prototypen, men alle designovervejelser er udført, og de mest vitale dele af prototypen er implementeret.

1.1 Projektforløbet og kommentare til teknologier

Rapporten og impementeringein blev udviklet over 4 uger. Af teknologier blev PHP valgt som implementeringssprog, PostgreSQL som database, og CVS som versioneringssystem. Alle teknoligerne har levet op til forventningerne.

PHP i version 4 har mange mangler når det kommer til objektorienteret programmering, det kan derfor ikke anbefales til implementering af størrer og mere advancerede webapplikationer, i alt fald ikke hvis man vil gøre meget ud af kodedesignet. PHP i version 5 bør dog have rettet op på mange af manglerne i version 4.

PostgrSQL er end en både advanceret og gratis RDBMS. Den kan stærkt anbefales som platform for implementering af databasedesigns. Som klient blev pgAdmin III til Windows brugt, programmet kan findes på `http://www.pgadmin.org/`.

Som CVS-klient blev Eclipse brugt der er et gratis Java-IDE udviklet af IBM. Det skal dog siges at at være CVS langt fra er Eclipses hovedformål, og at man derfor skal forvente at få en del mere med i købet. Eclipse kan hentes på `http://www.eclipse.org`

2 Kravspecifikation

Udarbejdelse af den endelige kravspecifikation vil ske i flere tempi. I første omgang vil den arbejdsgang, der ligger til grund for kravspecifikationen blive beskrevet. Udfra beskrivelsen af arbejdsgangen skal det være muligt at identificere de processer og elementer, der i sidste ende kommer til at udgøre fotoarkivet. Den information vil i første omgang komme til at definere kravspecifikationen. Herefter skal de enkelte punkter i kravspecifikationen gennemarbejdes således at de bliver mere implementeringsspecifikke.

2.1 Arbejdsgang

Der er en række brugere tilknyttet systemet. De falder alle inden for en eller flere af følgende kategorier:

Koordinator hvis hovedopgave er at oprette opgaver og uddelegere arbejde.

Fotograf hvis hovedopgave er at løse stillede opgaver.

Journalist der udelukkende leder efter billeder og ikke har skriveadgang til systemet.

Navnene er frit valgte. Det vigtige er deres funktion.

Hver fotograf tænkes at have sit eget lille område inde i systemet, hvor han/hun kan uploade billeder til. Dette område bruges til at klargøre billederne inden de sendes videre i systemet. Før billederne kan ses fra resten af systemet bliver fotografen nødt til at kategorisere dem (f.eks. ”Lejrbillede”), samt give dem en beskrivelse. Når de er færdigbehandlet og har fået tilknyttet den fornødne information, kan de tilknyttes til en opgave, hvis det er krævet.

En fotoopgave bliver stillet af en koordinator. En fotoopgave indeholder en beskrivelse, information om hvem der ejer den og hvem der skal løse den. Når en fotograf modtager opgaven kan han/hun tilknytte billeder til den. Fotoopgaven bliver på den måde et lille fælles arbejdsområde for fotografen og koordinatoren. De kan begge se og manipulere med de billeder og al anden data der ligger i opgaven.

Det skal være muligt at søge i billederne i arkivet. Alle billeder er blevet kategoriseret, og har fået tilknyttet en beskrivelse. Det er derfor muligt at søge dels ved at bladre igennem kategorier, men også vha. en mere generel fritekstsøgning.

Som et eksempel på hvorledes arbejdet forløber, er der udarbejdet følgende tænkte eksempel.

Koordinatoren opretter en ny opgave og tilknytter en beskrivelse. Han tilknytter evt. også anden data såsom en rutebeskrivelse til stedet, hvor billederne skal tages. Dette kan evt. være i form af en fil eller et fritekstfelt. Til sidst angiver han 2 fotografer, der skal løse opgaven.

De to fotografer kan nu se opgaven, når de logger ind (de kunne evt. få tilsendt en e-mail), de henter rutebeskrivelsen, tager ud til stedet, og tager billederne. Når de kommer hjem udvælger de deres billeder, uploader dem til fotoarkivet, tilknytter beskrivelse af de enkelte billeder og kategoriserer dem. Evt. lægges der mere end én version op af hvert billede, f.eks. en version retoucheret til tryk. Billederne er nu en del af fotoarkivet. Til sidst tilknytter de billederne til fotoopgaven, og supplerer

evt. med billeder, de har taget tidligere. Til sidst markerer de opgaven som værende løst.

Koordinatoren kan nu se at opgaven er markeret som løst. Hvis han er tilfreds med opgavens udførelse kan han melde opgaven endeligt løst.

2.2 Kravspecifikationen

Som det fremgår af workflowet er der en række krav som systemet skal leve op til. I det følgende afsnit vil kravene til systemet bliver præsenteret.

Det skal være muligt at:

- Fotograf
  - tilknytte metadata til billederne
  - tilknytte billedet til en kategori
  - lagre selve billedet i systemet
  - lagre flere versioner af det samme billede
- Koordinator
  - oprette fotoopgaver.
  - knytte fotografer til en opgave.
  - advisere en fotograf om en forestående opgave.
  - tilknytte filer til en opgave.
- Journalist
  - søge i metadataen tilknyttet billederne
  - hente et billede ud af arkivet
- Systemet
  - administrere brugere

2.3 Diskussion af kravspecifikation

Som det fremgår af kravspecifikationen er der en række problemstillinger, der skal tages stilling til. Der skal være en let tilgængelig brugergrænseflade, hvorigennem brugerne kan komme i kontakt med systemet og udføre deres ærinder. Derudover skal der være et sted at lagre data om billeder, filer og opgaver. Til slut skal der også findes en hensigtsmæssig metode til at lagre billeder og filer.

Selve brugergrænsefladen til fotoarkivet skal gøre det muligt for en koordinator at oprette en opgave. Til opgaven skal der kunne knyttes forskellig information samt filer, der vedrører opgaven. De fotografer, der skal udføre opgaven, vil være defineret i systemet og kan derfor, af koordinatoren, tilknyttes opgaven. Når opgaven er oprettet bliver fotograferne informeret om at der ligger en opgave til dem.

Når fotografer har modtaget opgaven og udført den, skal de have mulighed for at importere til arkivet. Når billederne importeres til arkivet, skal det være muligt at knytte diverse metadata til billederne og kategorisere dem. Herefter kan billeder knyttes til en opgave og lagres i arkivet. Før opgaven kan lukkes skal koordinatoren endeligt godkende opgaven. Desuden skal det kunne håndtere oprettelse af brugere og håndtering af deres rettigheder.

Brugergrænsefladen skal også stille funktionalitet til rådighed, der gør det muligt at søge i billederne og herefter eksportere dem, hvis det ønskes.

For at opnå størst mulige fleksibilitet er det blevet valgt at implementere brugergrænsefladen som et web-interface. Denne fremgangsmåde har visse fordele frem for en 'fed' klient. Ved at bruge et web-interface undgår man at brugeren skal have en installeret noget bestemt software på sin maskine udover en browser. Det gør også opdateringer af arkivet nemmere for administratoren. Selve implementering af brugergrænsefladen vil blive udført i PHP.

Al data om billeder, filer osv. vil blive gemt i en database. Her er valget faldet på PostgreSQL. Der vil blive udarbejdet en datamodel, som beskriver strukturen af databasen.

Det sidste store spørgsmål er om billeder og filer skal gemmes i databasen eller som filer på disken med en reference til dem i databasen. Der findes umiddelbart gode argumenter for begge dele.

En af fordelene ved at have filerne liggende på disken fremfor i databasen er, at der skal bruges færre I/O operationer når data skal fragtes mellem bruger-interfacet og arkivet. Ulempen ved denne måde er, at man mister de muligheder for kontrol, som en database tilbyder. Det er muligt selv at lave denne kontrol, men det er spørgsmålet, om forskellen i performance er så stor, at det kan retfærdiggøres. På grund af den manglende grundlag for at træffe en beslutningen på har vi valgt selv at udføre en test som en del af projektet.

3 Problemformulering

Udgangspunktet for problemformuleringen er det Det Danske Spejderkorps' nuværende måde at håndtere sit fotoarkiv på. For at give et indblik i hvordan problemet vedrørende lagring af fotos er opstået, vil der bliver givet en kort introduktion til problemerne.

Det Danske Spejderkorps har tilknyttet en række frivillige fotografer, der er organiseret i en fotogruppe. Disse fotografer tager imod fotoopgaver, der opstår i det daglige arbejde på korpskontoret. Fotograferne leverer materiale til selve arkivet, der består af digitale/digitaliserede billeder fra forskellige arrangementer. Ved hvert arrangement er der én eller flere fotografer, der sørger for at tage billeder. Disse billeder bliver indleveret på korpskontoret. Billederne bliver tilsidst indekseret og brændt ned på en CD. Derefter arkiveres de i et hængemappesystem. Dette kan også med tiden blive et problem, da cd'er er et forgængeligt medie.

Der er flere uhensigtmæssigheder ved den nuværende løsning. Blandt andet er det et problem, at der kun kan opnås adgang til fotoarkivet fra selve korpskontoret. Derudover foregår søgningen i billederne ved at bladre et ringbind med 'thumb nails' igennem. Når det ønskede billede er fundet, kan CD'en findes i arkivet og billedet kan tilvejebringes. Ydermere er det ikke muligt at oprette og uddelegere opgaver på en standardiseret måde. Det er heller ikke muligt at se status på de stillede opgaver.

De åbenlyse mangler i den nuværende måde at håndtere billeder på er ikke det store problem til hverdag. Men under den kommende landslejr, hvor ca. 20000 spejdere fra hele verden samles, vil antallet af opgaver, og dermed behovet for

koordinering, stige drastisk. Et estimat fra Det Danske Spejderkorps siger, at der under lejren vil arbejde 20-30 fotografer, som i perioden vil tage rundt regnet 2000 billeder hver. Et overslag fra Det Danske Spejdekorps lyder på at det vil resultere i ca. 200 GB data. Korpset har derfor ønsket at få implementeret et system der kan understøtte arbejdet.

3.1 Problemformulering

Vi ønsker at implementere en prototype af et system, der støtter processen hvor en koordinator og en gruppe fotografer har brug for at definere og løse en række foto-opgaver. Systemet skal lagre billeder, tilhørende metadata, samt anden form for data, der kunne være tilknyttet en foto-opgave.

Vi vil hovedsageligt fokusere på problemstillingerne forbundet med implementeringen af et sådant system, og udarbejde en fungerende prototype.

4 Lagring af filer

Et af målene for denne rapport er at finde en optimal måde at lagre billede- og projektfiler i fotoarkivet. Dette afsnit vil analysere de forskellige fordele og ulemper ved at gemme billeder og deres metadata i henholdsvis en Relationel Database (RDB) og filsystem. Efter analysen vil ydelsen af de forskellige modeller bliver afprøvet for at afklare hvilken, der egner sig bedst til fotoarkivet.

Den mest brugte måde at lagre filer på er naturligvis i et filsystem. Men alternative måder at lagre og tilgå filerne på kan vise sig at have fordele. Fotoarkivet har brug for at lagre to former for data: billeder og deres tilhørende metadata. I en relationel database er det særlig effektivt at relatere data, og måden disse relationer defineres på, er fastlagt. Et filsystem har derimod ikke nogen direkte understøttelse for arbitrær metadata. I praksis kan dette omgås ved at have en konvention for hvordan metadata lagres, som f.eks. en fil med samme fornavn med anderledes efternavn end den fil, der bliver beskrevet.

Relevante fordele ved at lagre filerne i et filsystem.

Hastighed Et filsystem vil som regel være optimeret til at kunne levere filerne så hurtigt som muligt. Ofte er filerne organiseret i en simpel og meget effektiv struktur, der tillader, at filerne kan findes hurtigt på disken.

Værktøjer Da lagring af filer i et filsystem langt fra er nogen nyhed, findes alle værktøjer, der skulle være nødvendigd for at tilgå filerne, såsom programmer til at kopiere, omdøbe og slette filer.

Understøttelse Der er f.eks. ikke noget problem i at tilgå filen igennem en webserver, da man kan forvente, at de fleste web-servere har understøttelse for filsystemer.

Relevante ulemper

Lagring af metadata Et filsystem vil som regel ikke tillade, at man lagrer brugerdefinerede metadata sammen med selve filen.

Manglende konsistens Det er som regel ikke muligt at definere afhængigheder i mellem individuelle filer i et filsystem. Dermed er det heller ikke muligt for

filsystemet at håndhæve eventuelle afhængigheder, der skulle findes. I sidste ende betyder det, at filsystemet alene ikke kan sikre at systemet forbliver i en konsistent tilstand. Dette kunne f.eks. blive et problem hvis man lagrer både billeder og deres metadata som filer. Systemet kunne i dette tilfælde bringes i en inkonsistent tilstand hvis man slettede billedefilen uden at slette metadatafilen.

Relevante fordele ved at lagre filerne i en relationel database.

Homogenitet Hvis filerne ligger sammen med metadata, er der kun ét sted man henter data fra. Systemet bliver dermed mere homogent, og det er ikke nødvendigt at inddrage funktioner til at tilgå filsystemer osv. Da metadata og filer er samlet, betyder det også, at man enten kan få fat i begge dele, eller ingen af dem. Hvis filerne lå for sig i et filsystem, kunne man forestille sig et scenarie, hvor filerne var utilgængelige, men metadataen stadig var tilgængelig. Man vil som regel gerne undgå disse såkaldte *patial failueres*.

Konsistents Eventuelle afhængigheder i mellem filerne og deres metadata bliver vedligeholdt af databasen. F.eks. kan man sørge for, at det ikke er muligt at oprette en tupel, i billed-relationen, før den tilhørende fil er indsat i en "filrelation".

Relevante ulemper ved at lagre filerne i en RDB.

Understøttelse Da filerne ikke bliver gemt i et "normalt" filsystem, kan man ikke forvente, at understøttelsen fra andre applikationer eksisterer. F.eks. vil man ikke kunne lade webserveren besvare en filforspørgsel på egen hånd. Men bliver nødt til at implementere et lille program, der kan agere som stedfortræder for filen, og hente den fra databasen.

Hastighed Alt efter implementeringen af den underliggende datastruktur som filerne vil blive gemt i, kan ydelsen af tilgangen til filerne være dårligere end for et normalt filsystem. En RDBMS arbejder som regel med betydeligt mindre blokke af data end de mange megabytes, som et billede kan fylde. Man kan derfor risikere, at en database, der ellers kan arbejde meget effektivt, bliver meget ineffektiv, når den skal flytte rundt på store blokke af data.

Det er ikke nyt at prøve at gemme binær data i databaser. De fleste RDBMS'er implementerer en eller anden form for understøttelse af større mængder binær data. PostgreSQL har to forskellige måder at håndtere problemet på.

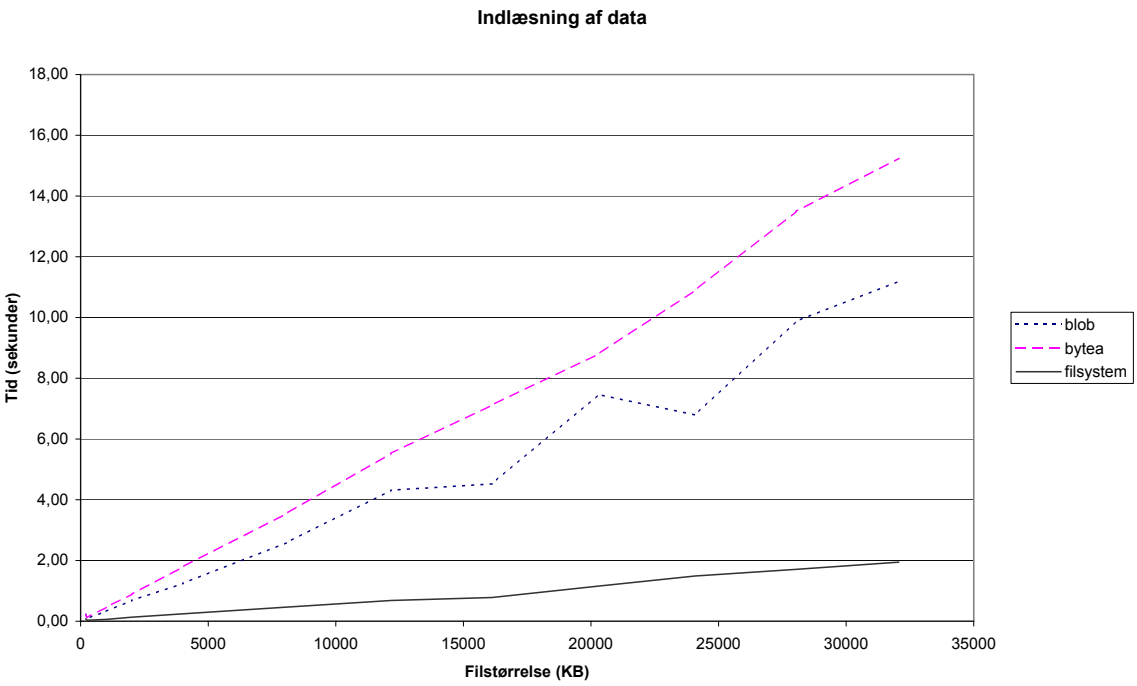
Den første er at behandle data som et såkaldt Binary Large Object (BLOB). PostgreSQLs implementering af dette hedder blot Large Objects(LO). De fleste andre RDBMS'er understøtter BLOBs i en eller anden grad. De forskellige RDBMS'er håndterer BLOBs på forskellige måder. I PostgreSQL har fokuset været at holde data under databasens kontrol, samtidig med at tillade god performance ved tilfældig tilgang til data. Det første mål bliver nået ved at lagre data i en tabel i databasen, i modsætning til at gemme hele filen i det bagvedliggende filsystem. Fordelen ved dette er, at man ikke kan tilgå filen "udenom" RDBMSen. Det andet mål bliver nået ved at dele filen op i en række blokke. Et indeks på tabellen gør, at man hurtigt kan finde en "tilfældig" blok. Det enkelte LO tilgås via en række specifikke funktioner. Kun to af disse kommandoer kan tilgås fra SQL, det drejer sig om `lo_import` og `lo_export` der henholdsvis opretter en ny BLOB ud fra en fil på disken, og skriver en BLOB ud i en fil på disken. De resterende kommandoer er tilgængelige fra klient-APIet, i dette tilfælde som PHP-funktioner. I stedet for at indsætte de enkelte large objects sammen med selve metadataen, indsætter man et såkaldt OID, der

er et id, der refererer til det enkelte LO. Der er ikke muligt at opsætte konsistensregler imellem en OID-attribut og det konkrete large object. Det betyder i sidste ende, at det godt kan lade sig gøre at slette billededata uden at gøre det samme med den tilhørende metadata.

Den anden måde at løse problemet er ved at benytte datatypen `bytea`, der beskriver en liste af bytes. En `bytea` attribut tilgås ligesom alle andre attributter i en relation. Dog kan binær data indeholde tegn, der er ulovlige ifølge SQL-syntaksen, så før data kan indsættes i databasen, skal disse tegn omskrives til deres lovlige repræsentation. På samme måde skal output-data fra databasen have rettet den lovlige repræsentation tilbage til original formen. Da bytea-attributten indgår i tabellerne på præcis samme måde som alle andre attributter, er det muligt at opsætte konsistens regler. Dermed er bytea *den eneste af de testede database-løsninger der sikrer konsistens*.

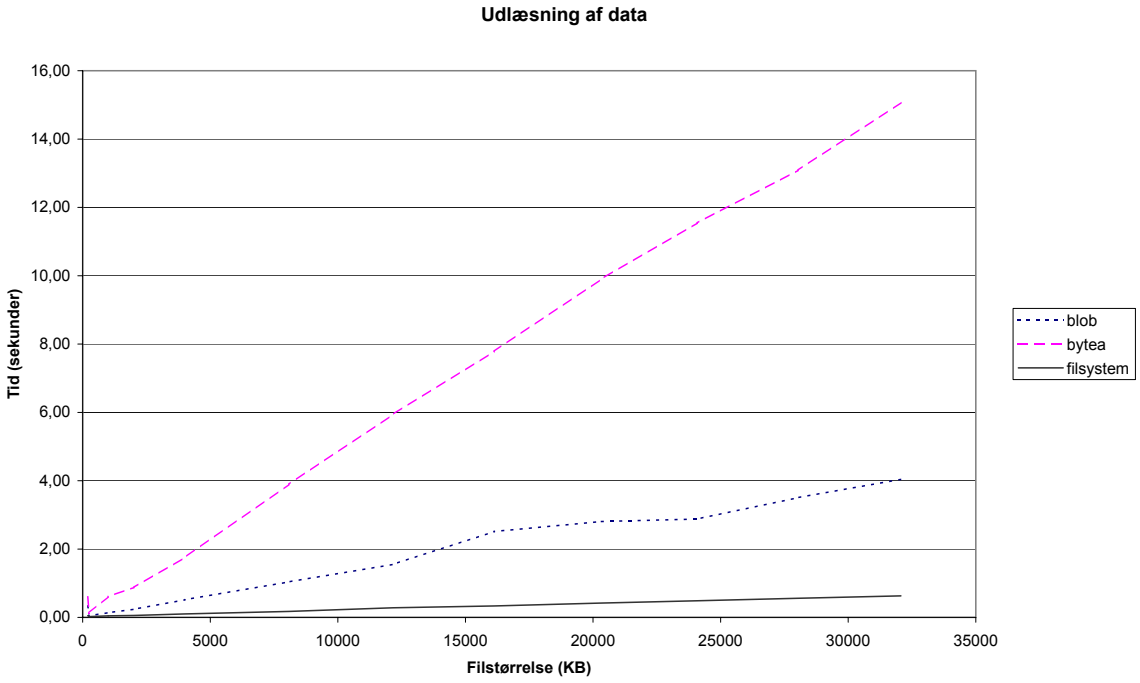
4.1 Afprøvning

Der vil nu blive planlagt og udført en test for at afgøre, hvor anvendelige de forskellige måder at lagre billederne på er. Der vil blive testet lagring i PostgreSQL som henholdsvis `Large Object` *og* `bytea`*, samt lagring direkte i filsystemet. De forskellige metoders karakteristik vil første blive gennemgået, og derefter vil der blive udført en afprøvning for at afklare deres svagheder og styrker.*



Figur 1: Indlæsning ved forskellige filstørrelser

Data vil meget sjældent blive opdateret i fotoarkivet. Ydermere vil der blive læst langt mere fra lagrings-systemet, end der vil blive skrevet. Det er derfor især interessant at undersøge lagringsystemernes ydelse, når der bliver læst fra dem. Dog vil indlæsningshastigheden også blive undersøgt, da den ikke må blive for høj. Brugeren vil normalt være vant til, at der kan gå et par sekunder fra han/hun klikker på



Figur 2: Udlæsning ved forskellige filstørrelser

et link til den næste side vises. Som udgangspunkt antages at brugeren vil forvente en reaktion fra systemet inden for 5 sekunder.

Man er helt nøjagtig interesseret i at undersøge følgende egenskaber:

Svarhastighed Hvor lang tid går der fra en forespørgsel efter en fil bliver sendt, til den første data bliver sendt?

Hastighed Med hvilken hastighed bliver data sendt?

Pladsforbrug Hvor meget ekstra plads bliver der brugt? Dette er især relevant for databaseløsningerne

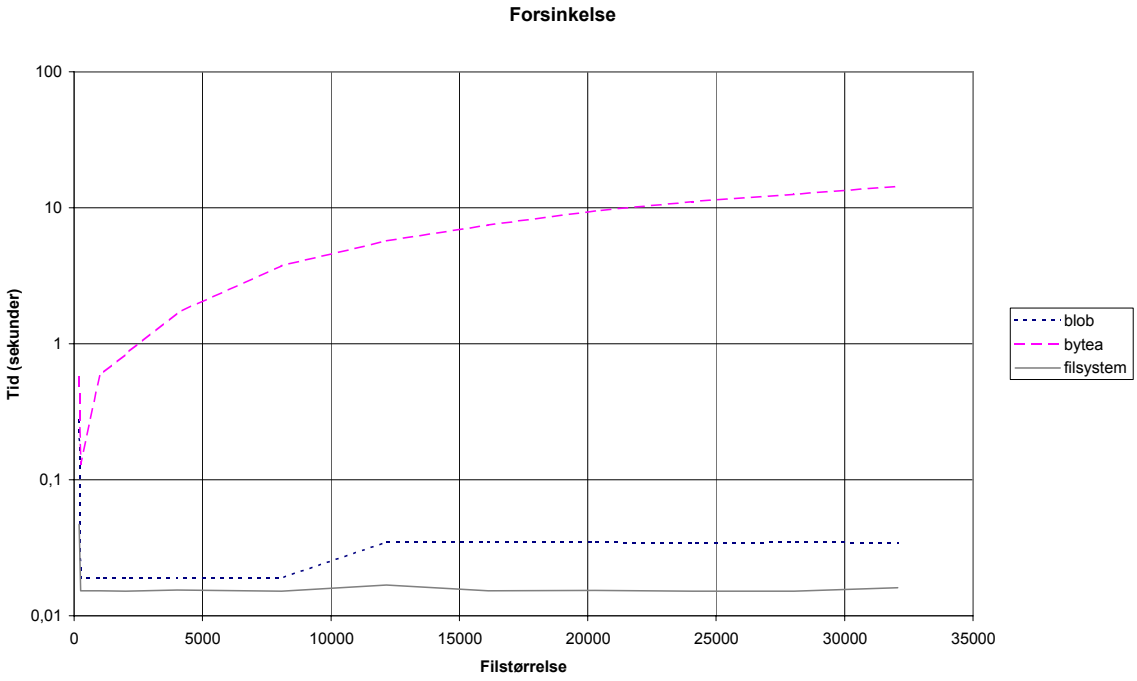
De forskellige løsninger har forskellige ulemper og fordele. Testens formål er at afklare hvor godt/slemt det står til, når løsningerne klarer sig værst.

Filsystem

Et filsystem er optimeret til at lagre filer, og må derfor antages at være det system, der klare selve den "rå" lagring af filer bedst. Filer i et filsystem er som regel indekseret efter det bibliotek, de ligger i. I værste fald vil filen blive fundet ved at lave en lineær søgning i biblioteket. I så fald må det antages, at der hvor filsystemet klarer sig dårligst, er når der er mange filer i samme bibliotek. Ydelsestabet forventes dog at være lineært.

Large Object

Large Objects deler filen op i en række blokke på hver 2KB. PostgreSQL har en central tabel, der bliver brugt til at gemme samtlige Large Objects. Det betyder, at ved lagring af f.eks. 200GB vil der blive oprettet 200 millioner tupler. Dette tal er kun afhængig af mængden af data, og ikke størrelsen af filerne. Databasen bruger



Figur 3: Reaktionstid ved forskellige filstørrelser, bemærk logaritmisk y-akse

et såkaldt B-træ som indeks-struktur[pgm]. B-træet garanterer at en given blok data kan findes på stort set konstant tid, selv for meget store mængder af data. Det kan derfor forventes, at filer på trods af det store antal af tupler i tabellen, kan udhentes på konstant tid.

Bytea

Ved at gemme data i selve tabellen, forøger man størrelsen af den enkelte tupel betydeligt. Så længe bytea-attributten ikke indgår i andre forespørgsler end lige netop den, der henter data til brugeren, burde det ikke have nogen indflydelse. Man kan ligesom med LO forvente, at data kan udhentes og indsættes i konstant tid.

4.2 Design af test

Det forventes, at systemet i værste fald skal kunne håndtere 200GB, og at størrelsen af filer vil fordele sig imellem 0-20 MB med overvægt i den lave ende. Det forventes, at de forskellige implementeringer opfører sig forskelligt alt efter størrelsen af de filer, de skal håndtere.

Databaser er som regel implementeret til at kunne lagre og udhente data effektivt selv ved skalering til meget store datamængder. Man kan derfor forvente, at ydelsen for både bytea og BLOB løsningen vil forholde sig konstant i forhold til mægden af data. Filsystemer kan som før nævnt have en svaghed, når der oprettes mange filer i samme bibliotek, men som det også er nævnt før, er der simple løsninger på det problem.

Målet med den første test er at afdække de grundliggende egenskaber for de forskellige løsninger. Alt efter hvordan bytea, blob og filsystems-implementeringen klare sig, vil der blive udført opfølgende test:

Test 1, filstørrelser

Formål: Afprøve ydelsen ved ind- og udlæsning af filer i forskellige størrelser.

Metode:

1. Indlæs en række filer i ca. størrelsen 0,2 - 0,5 - 1 - 2 - 4 - 8 - 16 - 32 MB. Mål tiden for hver indsættelse
2. Udlæs filerne. Notér hvornår den første blok data ankommer samt den samlede tid fra forspørgslen blev sendt til den sidste blok data er modtaget.

Resultatet af testen kan ses på figur 1, 2 og 3. PHP-implementeringen af testen samt den benyttede SQL-beskrivelse af tabellerne kan findes i bilag C.

Resultater

Ved både ind- og udlæsning er filsystems-løsningen klart hurtigere end BLOB og bytea. Bytea viser sig især at være dårlig til udlæsning, hvor den er ca 3-4 gange langsommer end BLOB og ca. 10 gange langsommere end filsystemet. Under testen var det nødvendigt at øge det maksimalt tilladte hukommelsesforbrug for PHP til 256 MB før bytea-løsningen kunne levere 32MB data. Årsagen er, at den mindste enhed PHP kan modtage fra databasen, er en tupel. Det er derfor nødvendigt at modtage *alle* 32MB, før de kan sendes ud til brugeren. Det kombineret med at data skal undersøges for eventuelle ulovlige tegn, der skal rettes tilbage til deres oprindelige repræsentation, gør at PHP skal jonglere med 32MB et stykke tid, før det kan sendes til brugen. Dette viser sig da også på figur 3 hvor man kan se, at størstedelen af tiden fra en forespørgsel er sendt til al data er modtaget, går med at vente på den første blok data. Alt i alt kan man konkludere, at bytea ikke egner sig til at gemme større mængder af binær data i denne sammenhæng.

BLOB klarer sig betydeligt dårligere end filsystems-løsningen ved indlæsning af data. Men da udlæsning, og ikke indlæsning har højeste prioritet i denne sammenhæng, er BLOB ikke umiddelbart udelukket af den grund. Ved udlæsning af data er BLOB adskillige gange langsommere end filsystems-løsningen, men præsterer dog stadig at levere 30MB data på under 4 sekunder. Da et LO er opdelt i en række sider, er det også muligt at sende data til brugeren efterhånden som denne kan håndtere den. Det betyder, at hukommelsesforbruget forbliver lavt. Alt i alt betyder det at BLOB strengt taget stadig ikke kan udelukkes som mulig løsning.

Det endelige valg falder dog ikke overraskende på filsystems-løsningen. Da BLOBs ikke tilbyder mere konsistens-garanti end filsystemet, er der ikke nogen reel fordel ved at bruge den i denne sammenhæng. Det er heller ikke så underligt, da LOs er implementeret for at give bedst mulig ydelse ved tilfældig adgang i filen, fremfor sekventiel adgang. Havde der derimod været krav til at tilfældige dele data i filerne skulle kunne tilgås hurtigst muligt, er det muligt at LOs ville have haft en fordel i forhold til filsystemet.

Yderligere tests

Der er ikke udført test af hvordan systemet klarer sig, når den samlede mængde af lagret data stiger. Grunden til dette er, at, når man ser bort fra bytea, er der ikke noget i implementeringen af hverken filsystemet eller BLOB, der burde få problemer, når mængden af data stiger. Data i BLOB er indekseret i et B-træ, der sikrer en konstant adgang til de enkelte tupler, og givet at man organiserer filerne lidt smartere end i et enkelt bibliotek, burde filsystemet heller ikke få problemer ved et højt antal af filer. Alt i alt antages det, at både BLOB og filsystemet vil skalere forholdsvist lineært.

5 Design

I dette afsnit vil designet af datamodellen, som implementeringen støtter sig op ad, blive beskrevet. Derefter vil der blive givet et overblik over designet af PHP-implementeringen af datamodellen. Datamodellen bliver baseret på den formulerede kravspecifikation og visualiseres vha. et E/R diagram. De efterfølgende afsnit vil gå i dybden med hvordan implementeringen benyttes.

Det skal bemærkes, at al implementering i PHP såvel som PostgreSQL er lavet på engelsk. I den følgene beskrivelse bliver de danske ord brugt for at lette læsningen, i appendix A findes en oversigt over de valgte oversættelser.

5.1 Design af datamodel

Datamodellen skal identificere entiteterne og relationere mellem disse. Det er gjort ved at tage udgangspunkt i kravspecfikationen. Resultatet kan ses på figur 4

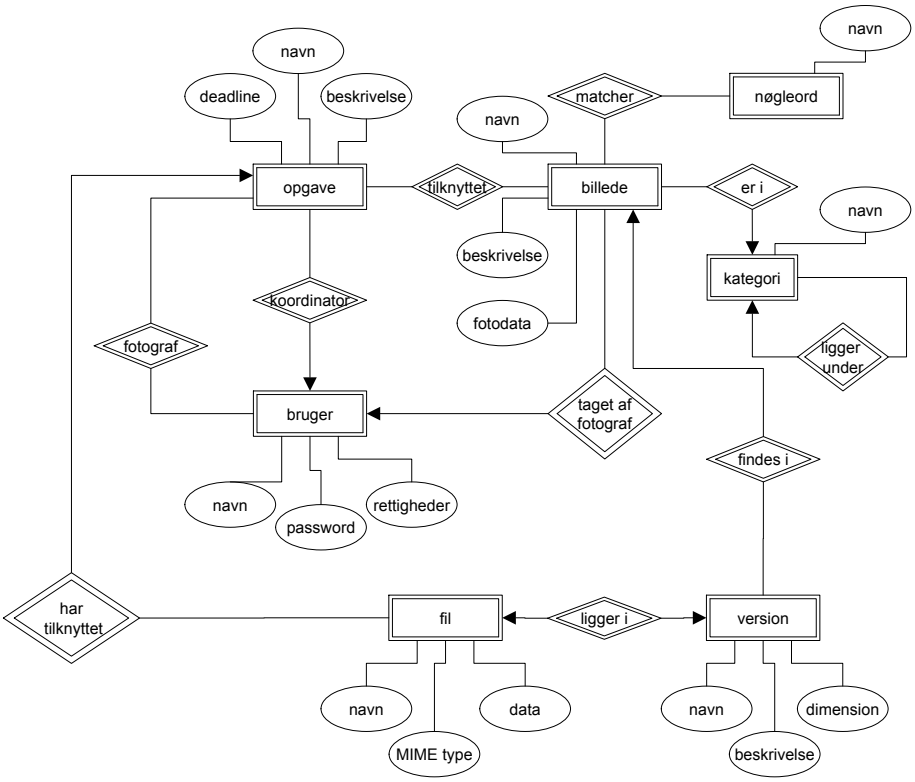
Et af de centrale elementer i datamodellen er entiteten billede. Entiteten repræsenterer et billede taget at en fotograf og ikke den konkrete binære data. Når billedet er importeret til arkivet, skal det have tilknyttet den information, som attributterne indikerer. Et billede skal således have tilknyttet et navn, en beskrivelse og derudover diverse foto data såsom lukkertid, brændevidde osv. Billedet skal derudover have tilknyttet en katagori fra én af de foruddefinerede kategorier. Dette er repræsenteret ved kategori entiteten. For at gøre det nemmere at søge i arkivet, skal billeder desuden have tilknyttet et antal nøgleord. Disse vælges også af fotografen fra en foruddefineret liste, der er repræsenteret ved nøgleord entiteten.

Det skal også være muligt at gemme et billede i flere versioner. Dette er repræsenteret ved versioner-entiteten. Hver version af billedet får tilknyttet et navn, en beskrivelse og desuden dimensionerne på billedet.

Når al denne information er fastlagt, kan filen arkiveres, hvilket er repræsenteret ved fil-entiteten.

En anden vigtig funktion, systemet skal tilbyde, er opgavestyring. Opgave entiteten repræsenter en opgave, oprettet af en koordinator. Koordinatoren findes, sammen med samtlige andre brugere, i en bruger entitet. Det er også her fotograferne, der skal tilknyttes opgaven, er oprettet

Både versioner og opgaver kan få tilknyttet filer. Disse filer er repræsenteret i entiteten fil.



Figur 4: E/R Diagram

Til forskel fra kravspecifikationen er der ikke nævnt en journalist i databasen, da denne blot vil være en bruger, der skal søge i systemet.

5.2 Design af PHP-implementering af datamodellen

Designet af PHP-implementeringen er i høj grad koblet sammen med designet af databasen idet API'et skal fungere som en abstraktion over databasen. Designet af APT'et kan ses på figur 5. Notationen i diagrammet er baseret på [Gra02]

De enkelte entiteter implementeres som PHP-klasser. Konkrete instanser af entiteterne kan dermed repæsenteres som instanser af de respektive klasser, og der kan tilgang til attribut-værdier kan kontroleres igennem 'getter' og 'setter' funktioner.

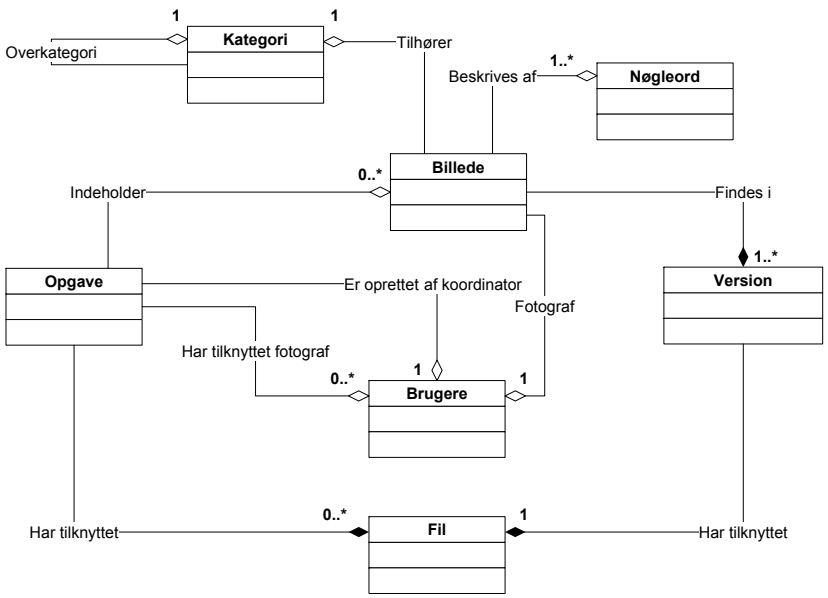
Udover de attributter, som entitet-klasserne har tilknyttet i kraft af den entitet de repræsenter (se figur 4), er relationerne imellem entiteterne også repræsenteret som referencer til instanser af andre entitetsklasser.

Følgende er en kort beskrivelse af de enkelte entitetsklasser, samt deres interne referencer.

Kategori Repræsenterer en enkelt kategori i hierakiet af kategorier. En kategori instans har et navn og en reference til sin overkategori i form af en kategoriintans. De "øverste" kategorier har en NULL-reference.

Nøgleord Repræsenterer en liste af nøgleord.

Billede Repræsenterer det tagne "billede", dvs ikke selve den binære data, men informationer som beskrivelse, lukkertid osv. Billede entiteten er en af de mest centrale dele af datamodellen, og har derfor mange tilknytninger til andre dele af modellen. En billedeinstans indeholder referencer til:



Figur 5: Klassediagram

- en Kategori-instans
- en Nøgleords-instans
- en fotograf i form af en Bruger-instans
- en liste af versions-instanser, der repæsenterer de versioner billedet findes i.

Opgave Repræsenterer en opgave. Opgaveinstansen indeholder en liste med nul eller flere Billed instanser, en liste med nul eller flere fotografer, en reference til præcis en koordinator, samt en liste af nul eller flere Fil instanser, der repræsenter evt. tilknyttede filer. Fotografer og koordinatoren er alle instanser af Bruger klassen.

Version Repræsenterer en version af et billede. Da Version i modsætning til Billede repræsentere et "fysisk" billede, har hver instans en reference til en instans af netop én Fil-instans.

Fil Repræsenterer en egentlig fil. En instans af denne klassse indeholder ingen referencer til andre objekter.

Dette lag kommer til at fungere som et API, der kan bruges til at tilgå datamodellen. Hovedopgaven for dette API-lag er defor at sørge for at opretholde og håndhæve de konsistens-regler, der er forbundet med datamodellen. Dette sikres ved udelukkende at kunne manipulere data i datamodellen igennem API-laget. Fordelen ved dette design er at man gør programørens opgave lettere, da han/hun ikke behøver at bekymre sig om at komme til at indsætte forkert data, da API-laget ikke vil tillade det. Man kunne f.eks. tænke sig at det ikke må være muligt at oprette et billede uden navn. I så fald er det API'ets opgave at sørge for, at en programmør ikke må kunne oprette et sådan billede. Ved at reducere indgangen til datamodellen til et enkelt sted, behøver data også kun at blive valideret et enkelt sted. Man vil dog kunne tillade, at der læses data uden om API'et. da kun skrivning til datamodellen kan bringe den i en ugyldig tilstand.

Formålet med APIet er at tilbyde en sikker grænseflade til datamodellen. Med sikker menes der, at en programmør, der bruger API'et, kan være sikker på at den

underliggende datamodel vil forblive i en gyldig tilstand så længe der kun benyttes funktionerne fra APIet til at manipulere den.

For at lette og generalisere implementeringen af de enkelte entiteter er der implementeret et framework i form af en fælles overklasse for de enkelte entitet-klasser. De generelle funktioner fra frameworket udgør sammen med funktionerne i de forskellige underklasser APIet til manipulation af datamodellen.

Frameworket håndterer følgende funktionalitet.

Instansiering af de enkelte entiteter.

Håndhævelse af konsistens regler imellem entitet-instanser.

Databasetilgang samt automatisering af generelle database-handlinger som opdatering og indsættelse af entitet instanser

Oprettelse og nedlægning af entitet-instanser.

Afhængigheder i mellem instanser af entiteter, således at det f.eks. ikke er muligt at slette en fil-instans før den version-instans, der refererer til den.

For at indgå i frameworket skal de forskellige underklasser som minimum implementere:

Konsistensregler i form af to funktioner, der skal kunne afgøre, om en konkret entitet instans vil bryde med datamodellens konsistensregler ved henholdsvis oprettelse og nedlæggelse af instansen.

Manipulation af instansens attribut-værdier

Afsnit 6.3 vil give en konkret beskrivelse af de funktioner, der er implementeret for at opfylde disse krav.

Ved at lave disse designvalg vil implementeringen blive mere advanceret end det måske kunne synes nødvendigt. Det ville kunne lade sig gøre at lave en implementering, der levede op til kravspecifikationen uden at skulle pakke alle entiter ind i klasser og lade disse klasser indgå i et større framework. En ulempe ved en 'mindre avanceret' implementering vil dog være at man hurtigt vil kunne miste overblikket, og at den enkelte programmør derfor ikke vil kunne føle sig nær så sikker på, at en given entitet f.eks. var implementeret korrekt. Ved at have regler for, hvordan en entitet skal implementeres (som en klasse) og hvordan den skal indgå i systemet (dvs frameworket), gør man designet mere komplekst, men den egentlige implementering af de enkelte dele langt simplere.

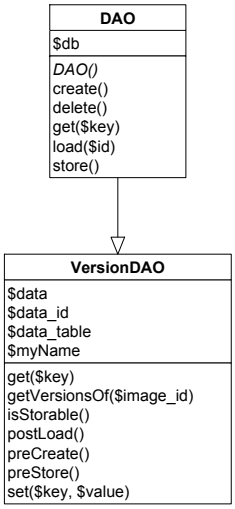
6 Implementering

Det centrale i implementeringen af datamodellen er, udover den egentlige implementering i PostgreSQL, de PHP-klasser, der er implementeret som et API til datamodellen. Dette afsnit vil beskrive grundideen bag implementeringen af disse klasser, samt give et konkret eksempel på, hvordan entiteten version er implementeret i klassen VersionDAO. Se evt. den tekniske brugervejledning i afsnit 7.2 på side 26 for en demonstration af hvordan APIet skal anvendes.

6.1 Implementering af datamodellen i PostgreSQL

Da implementeringen af E/R diagrammet på figur 4 i en relationel database er relativ ligefrem, vil der ikke blive givet en nærmere beskrivelse heraf. SQL beskrivelse af tabellerne kan findes i bilag B. De eneste konsistens-regler, der er opsat, er regler om fremmednøgler i mellem tabellerne. De resterende konsistens-regler bliver håndhævet af PHP-laget, der bliver beskrevet i det følgende.

6.2 PHP-implementering af datamodellen



Figur 6: Klassediagram over DAO klassen og underklassen VersionDAO

6.3 Entitetframeworket DAO

Frameworket tilbyder en række generelle funktioner til at manipulere de enkelte entiteter. Jævnfør evt. API dokumentationen i bilag E

DAO() konstruktør, der sørger for at initialisere frameworket. Underklasser skal sørge for, at denne bliver kaldt.

create() bruges når man ønsker at oprette en ny instans af en entitet i databasen. Dette gøres i praksis ved først at sætte alle attributter for instansen og derefter kalde create(). Før entiteten bliver oprettet, vil funktionen isStorable() som underklasser skal implementere først blive konsulteret. Se evt. beskrivelsen af VersionDAO implementeringen.

delete() sletter en instans af en entitet. Før entiteten bliver slettet vil funktionen isDisposable() som underklasser skal implementere først blive konsulteret. Se evt. beskrivelsen af VersionDAO implementeringen.

get(attribut) en generel implementering, der giver læse-adgang til attributter i entiteten. Denne funktion bør implementeres af underklasser hvis der ønskes kontrol over hvordan værdierne bliver læst. Dette er f.eks. tilfældet i VersionDAO, der beskrives i næste afsnit.

load(primærnøgle) indlæser entitets instans fra databasen.

store lagre en entitets tilstand i databasen. Før entiteten lagres konsulteres funktionen isStorable() som underklassen skal implementere.

En underklasse af DAO skal som minimum implementere funktionerne.

isStorable() der skal returnere en boolesk værdi, der angiver om den pågældende instans af entiteten overholder eventuelle konsistens-regler og dermed er lovlig at gemme.

isDisposable() der skal returnere en boolesk værdi, der angiver om alle andre instanser af entiteter er uafhængig af den pågældende instans af entiteten og dermed om den er lovlig at fjerne.

set(attribut, værdi) der skal give adgang til at ændre værdier i entiteten. Grunden til at denne funktionalitet ikke håndteres generelt er, at den enkelte implementering af entiteten bør sikre at ændringerne ikke bringer entiteten i en ulovlig tilstand.

Disser funktioner er også implementeret i DAO-klassen. Grunden er at PHP i version 4 ikke understøtter abstrakte funktioner. I mangel af bedre er funktionerne derfor implementeret i DAO-klassen, men vil stoppe eksekveringen af php-scriptet´, hvis de bliver kaldt. Da `isStorable()` f.eks. altid vil blive kaldt, når entiteten bliver oprettet første gang, betyder det effektivt set at underklassen vil være ubrugelig indtil DAOs implementering af `isStorable()` bliver overskrevet.

En underklasse skal definere felterne

data_table angiver navnet på den databasetabel hvori data for instanser af entiteten kan findes.

data_id angiver hvilken attribut i tabellen der er primær nøgle for entitet instansen.

data et associativt array der har navnene på tabel attributterne som nøgler, og eventuelle standard værdier som værdier.

myName navnet på entiteten. Bruges udelukkende til fejlfindings information.

Derudover kan følgende funktioner overskrives, hvis den enkelte implementering har brug for mere kontrol med ind og udlæsning af data.

postLoad() kaldes umiddelbart efter data er blevet indlæst fra databasen. Dette tillader f.eks. at implementeringen instansierer andre entiteter ud fra de eventuelle fremmednøgler, der er lagret i databasetabellen.

preCreate() kaldes umiddelbart før isStorable() og før en nyoprettet entitet bliver lagret. Dette tillader at implementeringen forbereder sig på at blive lagret for første gang ved f.eks. at fremskaffe en primær nøgle for den nye entitet.

preStore() kaldes umiddelbart før en eksisterende entitets attributter lagres tilbage i databasen. Dette tillader f.eks. implementeringen at udhente fremmedenøgler fra eventuelle instanser af andre entiteter.

6.4 Implementering af entiteten version

For at give et indtryk af, hvordan entiteter kan implementeres i frameworket vil der nu blive givet en beskrivelse af hvordan entiteten version er implementeret i klassen VersionDAO. Der bliver givet en beskrivelse af samtlige funktioner implementeret i klassen, påkrævede såvel som ekstra funktioner. En oversigt over hvordan de forskellige funktioner og felter er fordelt imellem DAO og VersionDAO kan ses på klassediagrammet på figur 6 på side 21.

Entiteten version bliver implementeret således at den skal have netop én reference til en fil-instans. Version har også en én til mange relation til billede. Denne relation kan implementeres enten ved at hver billede-instans har en liste af versions instanser, eller ved at hver versions-instans har en reference til sit billede. Det kan i princippet også lade sig gøre at implementere begge regler på én gang. I dette tilfælde er den første løsning dog valgt. Versions-instanser er derfor gyldige selvom de ikke har en reference til en billede-instans.

Alt i alt bør implementeringen af version-entiteten sikre, at der altid er en gyldig reference til en fil instans. Derudover er det også et krav at en version har et navn.

Følgende funktioner er implementeret i VeresionDAO.

get(attribut) returnerer attributterne tilknyttet entiteten. Derudover vil Fil og Billedinstanserne tilknyttet versionen blive returneret ved forespørgsler efter henholdsvis 'file' og 'image'.

isDisposable() Undersøger om Versionen er tilknyttet et billede. I så fald returneres false da billedet skal slettes før versionen.

isStorable() Undersøger om instansen har tilknyttet en fil, og om attributten name er sat.

postLoad() indlæser den tilknyttede Billede og Fil instans.

preCreate() udhenter en ny primærnøgle fra en sekvens i databasen og sætter id attributten.

preStore() udhenter primærnøglerne fra den tilknyttede Fil og Billede instans og sætter attributterne file_id og image_id.

set(attribut, værdi) sætter attribut-værdierne for entiteten. Implementeringen tillader ikke at hverken file_id eller image_id bliver sat manuelt. Derimod tager funktionen imod attributterne 'file' og 'image', som gemmes i interne felter. Iderne fra disse instanser bliver senere gemt ned i databasen af preStore().

Udover de 'normale' funktioner implementerer VersionDAO også

getVersionOf(primærnøgle) der er en statisk metode der bruges til at udhente en liste af Version instanser tilknyttet et billede.

Denne funktion kunne godt være implementeret i en sideliggende fil da den er statisk og ikke har noget egentligt behov for at ligge i VersionDAO. Det er er dog på den anden side logisk at lægge funktioner der behandler versions-instanser i VersionDAO.

6.5 Den faktiske implementering

Grundet tidsmangel er ikke alle entiteter blevet implementeret. De entiteter der er blevet implementeret er.

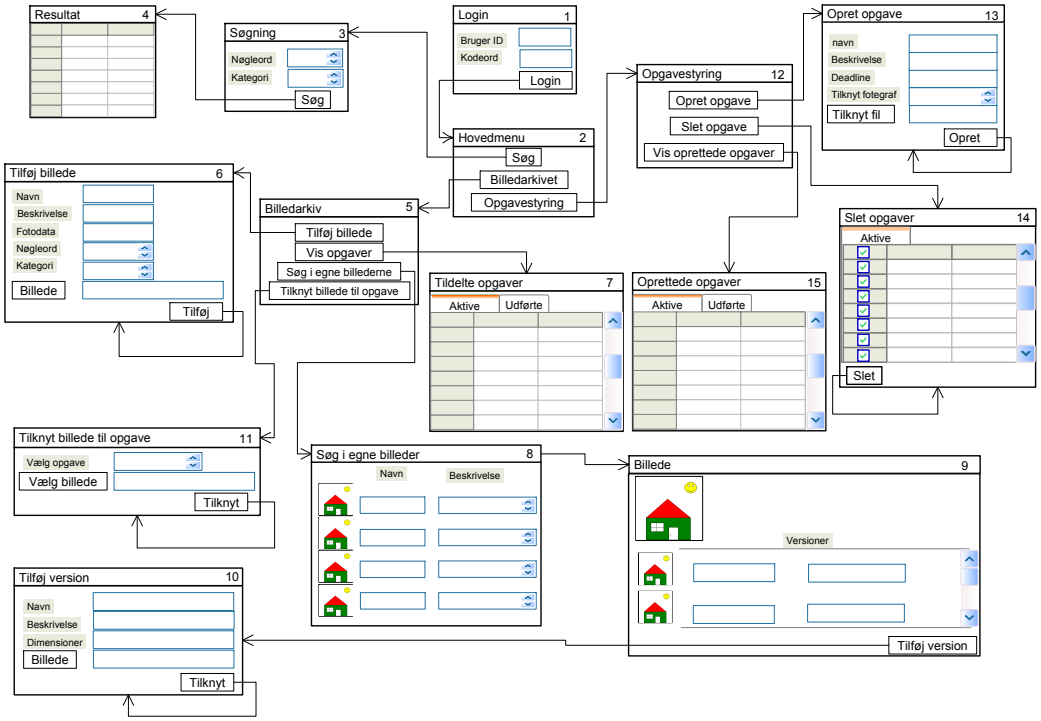
- Kategori
- Billede
- Version
- Fil

Disse entiteter er blevet valgt fordi de tilsammen er nok til at demonsterere alle funktionaliteter i frameworket. For at lette udskrivning af fejlfindingsinformation og oprettelse af forbindelse til databasen er der derudover indtaget kode til håndtering af dette. Det drejer sig konkret om filerne `SSLog.class.php` samt `Core.class.php`,

der kan findes i API-dokumentationen i bilag E. Til databaseinteraktion benyttes desuden PEAR-implementeringen af et databaseabstraktionslag. [PEA]

7 Brugervejledning

Dette afsnit er en brugervejledning til hvordan det implementerede system er tænkt brugt. Brugervejledningen er delt op i to afsnit der begge baserer sig på et sitemap. Første afsnit er tiltænkt brugeren af programmet, der skal kune benytte det i arbejdsmæssig sammenhæng. Det næste afsnit er tiltænkt programmøren der skal implementere den bagvedliggende logik, og som bruger APIet.



Figur 7: Sitemap

7.1 Struktur i sitet

Design af sitet skal tage udgangspunkt i figur 7. Dette oversigtskort er en visuel beskrivelse af hvorledes brugergrænsefladen hænger sammen. Det skal sætte en bruger i stand til at forstå hvorledes brugergrænsefladen er struktureret. Endvidere skal det beskrive den funktionalitet hver enkelt side stiller til rådighed for brugeren. Det grafiske design vil ikke blive gennemgået.

Til oversigtskortet skal det bemærkes at knapper uden pile på åbner en fil browser.

Design vil blive gennemgået side for side. På hver side er der et tal i øverste højre hjørne. Dette tal er ikke en del af selve designet, men vil blive brugt til at at referere til siderne.

1 - Login Dette skærmbillede skal modtage et bruger id og en kode fra en bruger. Når der klikkes på Login knappen skal det undersøges, om bruger eksisterer og i så fald hvilke rettigheder brugeren har

2 - Hovedmenu Herfra skal det være muligt for brugeren at få adgang til de dele af fotoarkivet, som hans login tillader. De enkelte knapper skal åbne et nyt skærmbillede, der svarer til teksten på knappen

3 - Søgning Dette er det resulterende skærmbillede fra knappen søg i hovedmenuen. Brugeren skal kunne søge ved at udvælge kategoier og nøgleord. Der skal evt. senere også kunne søges i beskrivelsen af billedet. Et klik på knappen søg skal åbne et vindue med resultatet af søgningen

4 - Resultat Her skal resultatet af søgningen fra menu 3 vises

5 - Billedarkiv Dette billede giver adgang til alle de funktioner, der vedrører håndtering af billeder. Klik på de forskellige knapper skal åbne nye skærmbilleder svarende til knappens tekst

6 - Tilføj billede Mulighed for at tilføje et billede til arkivet skal håndteres fra dette skærmbillede. Brugeren skal udfylde samtlige felter med værdier. Valg af billede foregår ved et klik på Billede-knappen, der skal åbne en filbrowser, hvorfra der kan vælges et billede. Når der klikkes på knappen Tilføj undersøges det om alle felter er udfyldt korrekt og dernæst kan billedet lagres i arkivet

7 - Tildelte opgaver Skærmbilledet viser en oversigt over den pågældende fotografs opgaver udførte, såvel som aktive

8 - Søg i egne billeder Her skal fotografer kunne få en liste over de billeder der er gemt i hans del af arkivet. Ved at dobbelt-klikke på et af billederne vil han blive sendt videre til billede 9

9 - Billede Her vises et billede sammen med alle de versioner billedet er gemt i. Det skal være muligt at tilføje endnu en version af billedet ved at klikke på Tilføj version knappen.

10 - Tilføj version Her skal specificeres navn, beskrivelse og dimensioner på den version man ønsker at tilføje. Derefter skal selve filen, der skal tilføjes, vælges via en filbrowser.

11 - Tilknyt billede til opgave Her skal det være muligt at tilknytte et billede til en opgave. Ved vælge en aktiv opgave fra listen og dernæst vælge et billede, der skal tilknyttes. Knappen tilknyt sørger for at knytte de 2 sammen

12 - Opgavestyring Dette er koordinatorens hovedmenu. Herfra skal det være muligt at styre opgavedelen af fotoarkivet.

13 Opret opgave Dette skærmbillede opretter en ny opgave. En opgave skal have et navn, en beskrivelse og en dead line. Derudover skal den have tilknyttet et antal fotografer, der skal løse opgaven. Disse vælges fra en liste af oprettede fotografer i systemet. Det skal også være muligt at tilknytte filer til opgave via en fil browser. Dette gøres ved et klik på Tilknyt fil knappen. Ved at klikke på Opret knappen bliver opgaven oprettet og de personer, der skal løse opgaven skal adviseres.

14 - Slet opgave På dette billede skal en koordinator have mulighed for at slette oprettede aktive opgaver. Der skal vises en liste over opgaver koordinatoren har kontrol over. Han skal have mulighed for at markere dem han ønsker slettet. Et klik på knappen Slet skal derefter slette de valgte opgaver

15 - Oprettede opgaver Skærmbilledet viser en oversigt over den pågældende koordinators opgaver udførte såvel som aktive

7.2 Brug af APIet

Der vil nu blive givet en demonstration af hvordan APIet kan bruges til at implementere de dele af sitemappet det interagere med datamodellen. Da det kun er entiteterne Billede, Version, Fil og Kategori der er implementeret kan der ikke gives forslag til implementeringen af alle sider i sitemappet. Der vil her blive givet forslag til hvordan de siderne 3, 4, 6, 8, 9 og 10 kan implementeres.

3 - Søgning Da en søgning er læsning fra databasen, kan listen af resultater udtrækkes via en dirkete SQL-forspørgsel uden om APIet.

```
$db = Core::getDB();
$result = $db->("SELECT id,name,parent FROM category");
$versions = array();
while($row = $result->fetchRow()){
    $versions[] = $row;
}
```

Listen $version vil nu indholde primærnøgler og navne på alle kategorier. Evt bør man nok gøre noget for at opstille kategorierne således at brugeren kan se den hierakiske struktur.

4 - Resultat Det antages at værdien primærnøglen for den kategori der blev valgt på den forgående side er lagret i variablen $category_id.

```
$db = Core::getDB();
$result = $db->query("SELECT image.id FROM image
                        category_id = $category_id");

$images = array();
while($row = $result->fetchRow()){
    $image = new ImageDAO()
    if($image->load($row['id'])){
        $images[] = $image;
    }
}
```

Herefter vil listen $images instanser af alle billeder der lå i kategorien med id $category_id.

Når nøgleord er implementeret skal denne kode dog udvides således at der kun udtages billeder der ligger i den rigtige kategori og har tilknyttet de rigtige nøgleord.

6 - Tilføj billede Det antages at `$filepath` indeholder den absolutte sti til filen der er blevet uploaded og at `$filename` indeholder filnavnet.

```
$file = new FileDAO();
$file->set('name', $filename);
$file->set("data", $filepath);
$file->create();

$version = new VersionDAO();
$version->set('file', $file);
$version->set('name', "Original");
```

```
$version->create();

$image = new ImageDAO();
$image->set('name', $imagename);
$image->set('category', $category);
$image->create();
$image->addVersion($version);
$image->store();
```

8 - Søg i egne billeder Det antages at brugeren er logget ind og at brugerens id kan findes i variablen `$userid`

```
$db = Core::getDB();
$result = $db->query("SELECT id FROM image
                        WHERE photographer_id = $userid");
$images = array();
while($row = $result->fetchRow()){
    $image = new ImageDAO();
    if($image->load($row['id'])){
        $images[] = $image;
    }
}
```

9 - Billede Det antages at primærnøglen for billedet, der skal vises kan, findes i variablen `$image_id`.

```
$image = new ImageDAO();
$image->load($image_id);

$versions = $image->getVersions();
```

10 - Tilføj version Det antages at `$filepath` indeholder den absolutte sti til filen der er blevet uploaded og at `$filename` indeholder filnavnet.

```
$file = new FileDAO();
$file->set('name', $filename);
$file->set("data", $filepath);
$file->create();

$version = new VersionDAO();
$version->set('file', $file);
$version->set('name', "Original");
$version->create();
```

8 Afprøvning

Formålet med afprøvningen er at afklare om implementeringen lever op til de krav, der er stillet i kravspecifikationen. Derudover skal testen afgøre om systemet forbliver i en gyldig tilstand uanset hvilke input data, der kommer fra brugeren.

Den første del af afprøvningen er en analyse, hvor de afprøvnings-relevante dele af systemet bliver identificeret.

8.1 Afprøvnings overvejelser

Selve afprøvningen vil fokusere på den del på den af fotoarkivet, der vedrører indsættelse og versionering af billeder. Afprøvningen kommer således ikke til at omfatte hele APIet da kun dele af det er implementeret.

Følgende funktionalitet vil blive testet

- tilknytte metadata til billederne
- tilknytte billedet til en kategori
- lagre selve billedet i systemet
- lagre flere versioner af det samme billede

Når disse krav er afprøvet tilfredsstillende, lever implementeringen op til kravspecifikationen. Dette vil reelt være en blackboxtest af API-laget. En afprøvning af dette lag alene vil ikke garantere korrektheden af implementeringen, da API-laget bygger på en underliggende implementering af datamodellen. En nærmere unit-test af det underliggende lag vil kunne hjælpe til at afdække eventuelle fejl og mangler i implementeringen. Dette afsnit vil holde sig til at beskrive implementeringen og afprøvning af blackboxtesten, da både en unittest og en blackboxtest vil være for omfattende for denne rapport. Blackboxtesten giver i så fald det bedste billede af om implementeringen lever op til kravspecifikationen.

Afprøvningen har til formål at undersøge de grænsetilfælde, der vil være for input af data. Desuden skal afprøvningen undersøge, om de regler der gælder i datamodellen også er overholdt i API-laget. En af de regler, der skal være overholdt, er eksempelvis at et billede skal have et navn. Det betyder at bruger skal blive gjort opmærksom på at der er en fejl, hvis han forsøger at tilføje et billede uden navn. Det er denne type fejl blackboxtesten fokuserer på.

8.2 Analyse af afprøvning af API-laget

Afprøvningen af API-laget vil beskæftige sig med den liste af funktionaliteter, der er beskrivet i afsnit 8.1. Det enkelte område vil blive behandlet hver for sig.

Den først del af API'et, der skal afprøves, er oprettelsen af et nyt billede. Før et billede kan accepteres af systemet skal det have følgende data tilknyttet.

- et navn
- en beskrivelse
- en kategori
- et nøgleord
- binær billeddata

Alle disse data om et billede skal være til stede før billedet kan lagres i databasen. Når den første version af et billede indsættes i arkivet er det ikke muligt for brugeren at indtaste versionsdata. Dette er kun nødvendigt for yderligere versioner af billedet.

Det er et krav fra databasen, at alle værdier for de enkelte data elementer bliver udfyldt. Der er i øjeblikket ikke noget til hvilke værdier de udfyldes med. En bruger kan således selv bestemme over hvordan felterne bliver udfyldt. Dette gælder naturligvis ikke for de foruddefinerede data som nøgleord og kategori.

Når et billede er gemt i arkivet skal det være muligt at tilføje flere forskellige versioner af det originale foto. Udover den generelle data for fotoet, som er lagret med det originale foto, skal hver ny version have tilknyttet data, der er unik for de enkelte versioner.

- en version skal have et navn
- en version skal have en beskrivelse
- diverse dimensioner på billedet skal indtastes.

Alle data for vesion skal også være udfyldt for at ny version kan accepteres af arkivet. Den konsistente indebærer at alt data er sat korrekt via `set` funktionen. Hvorimod den inkonsistente vil forsøge at kalde create på et objekt med manglende referencer eller ikke satte datafelter.

8.3 Test af API'et

For at undersøge om API'et lever op til de stillede krav, skal funktionerne, der muliggør oprettelse af et billede om en version, afprøves.

Generelt skal funktionerne afprøves med en konsistent og en inkonsistent tilstand. Den konsistente indebærer eksempelvis at al data, der skal ind i systemet eller data man forsøger at hente ud, findes. Modsat vil inkonsistente tilstand eksempelvis forsøge oprette objekter uden den fornødne data tilgængelig eller opdatere data, der ikke findes. Hver funktion vil i det følgende blive testet med begge tilstande. Testen udførelse vil blive beskrevet nærmere i det følgende.

- Den første test vil omhandle oprettelse af objekter via `create` funktionen.

Før et billede kan oprettes, skal der være referencer til en kategori og en version. Derudover skal attributterne navn og beskrivelse være sat. Første vil testen forsøge at oprette et objekt hvor alle data er til stede. Dernæst forsøges objektet oprettet med manglende data. Der vil blive foretaget en udtømmende test af manglende data

For at oprette en kategori kræves det, at kategorien har et navn. Testen vil oprette en kategori med navn og en kategori uden navn.

Testen af version skal oprette et versionsobjekt med en reference til en fil og attributten navn. Herefter oprettes objektet i hvor enten fil referencen eller attributten mangler.

Den sidste test vedrører oprettelse af et Fil objekt. En fil skal have en navneattribut og en reference til data. Fil vil blive forsøgt oprettet med attributten og referencen tilstede. Derfor vil fil blive forsøgt oprettet hvor enten attributten eller referencen mangler.

- Den næste test skal afprøve funktionen `set`. Testen vil blive udført ved at `set` funktionen vil blive kaldt på de 4 objekter, nævnt ovenfor, med de korrekt data. Herefter vil `set` blive kaldt med data, der ikke er konsistent.

- Funktionen `get` vil blive testet på samtlige objekter ved at prøve at få fat i de enkelte attributter og referencer på objektet. Dernæst vil `get` blive forsøgt udført på data, der ikke eksisterer.
- Når data skal hentes ud af database skal funktionen `load` kaldes. Funktionen vil hente data ind fra hver at de 4 objekter. Herefter vil det blive forsøgt at hente data ind fra 4 objekter der ikke eksisterer.

Opdateringen af et objekt foregår ved først at loade et objekt. Derefter kan man via `set` opdatere objektet. For at gemme det igen kaldes `store` på objektet. Funktionen `store` vil ikke blive testet, da den ikke vil kunne bringe et objekt på inkonsistent form, da funktionerne, der opdaterer objektet, i sig selv ikke kan bringe objektet i en inkonsistent form.

- Til slut vil `delete` blive afprøvet. Denne funktion skal forsøge at slette data fra arkivet.

8.4 Implementering af tests

Implementering af test vil teste hver af de 4 objekter i henhold til beskrivelsen i 8.3. Til hver test vil der høre en attributtest. Test vil forsøge at indlæse data til en objekt instans, og herefter hente de enkelte attribut værdier ud. Yderligere skal der oprettes en instans af det pågældende objekt og forsøge at sætte alle værdierne med gyldige værdier. Herefter skal objekterne forsøges oprettet med ugyldige værdier. Man kan eksempelvis forsøge at oprette et billede uden et navn. I test afsnittet vil der være eksempler på dette, men da manipulation af attributter er implementeret således at de er uafhænginge af hinanden, vil samtlige kombinationer dog ikke blive beskrevet, da dette regnes for trivielt. I stedet vil de særlige forhold for hvert objekt blive fremhævet.

8.5 Testeksempel

Filtesten vil være repræsentativ for hvordan oprettelsen af et objekt og attribut testen skal udføres. Bemærk at implementeringen af Fil-entiteten vil importere filen 'test.jpg' fra filsystemet hvis man kalder `set('data', 'test.jpg')`, imens `set('name', 'test.jpg')` blot vil sætte den pågældende attribut i entiteten.

Handling	Forventet resultat
Lav en fil instans	OK
Sæt navn til 'test.jpg'	OK
Sæt data til 'test.jpg'	OK
Gem instansen	OK

Handling	Forventet resultat
Lav en fil instans	OK
Undlad at sætte navn	–
Sæt data til 'test.jpg'	OK
Gem instansen	Fejl

Den samme type test skal udføres for de andre objekter, men med forskellige kombinationer af undladte attributter. Der vil skulle sættes en reference i stedet for en attribut. Det vil f.eks. ske i version.

Handling	Forventet resultat
Lav en Fil instans	OK
Sæt navn til 'test.jpg'	OK
Sæt data til 'test.jpg'	OK
Lav en Version instans	OK
Sæt navn til 'version 1'	OK
Sæt fil til Fil-instansen	OK
Gem instansen	OK

Selve oprettelsen af de forskellige objekter skal ske i en bestem rækkefølge. Nærmere bestemt skal de objekter, et billede skal referere til, være til stede før billedet kan oprettes. Endvidere har version objekt et krav om at der skal være et fil objekt tilstede før en version kan oprettes. Det eneste krav til kategori er at den bliver oprettet før billedet bliver oprettet. Dvs. at oprettelse af et billede forudsætter følgende rækkefølge

- opret kategori instans
- opret fil instans
- opret versions instans
- opret billede instans

En test af dette vil indebære at man prøver at oprette objekterne i en rækkefølge, der vil lede til inkonsistente objekter. Eksempelvis at oprette et billede for kategorien, som skal tilknyttes, findes.

Afhængigheden i billedinstansen for også konsekvensen, hvis man vil prøve at slette data fra databasen. Således må det ikke være muligt at slette en data, der er knyttet til et billede, med mindre billedet først er slettet. Testen for dette skal derfor oprette et billede og derefter forsøge at slette data tilknyttet billedet. Endvidere skal man ikke kunne slette en fil før versionen der peger på filen er slettet. En test af dette kunne udføres således

Handling	Forventet resultat
Opret billede	OK
Slet fil	Fejl
Slet version	Fejl
Slet kategori	Fejl
Slet billede	OK
Slet fil	Fejl
Slet version	OK
Slet kategori	OK
Slet fil	OK

Eksempel på implementering:

```
$file = new FileDAO();
$this->assertTrue($file->set('name', 'test.jpg'));
$this->assertTrue($file->set('data', 'd:/test.jpg'));

$version = new VersionDAO();
$this->assertTrue($version->set('file', $file));
$this->assertTrue($version->set('name', 'version 1'));
$this->assertTrue(($version->create());
```

```
$this->assertFalse($file->delete());
```

8.6 Konklusion

Under testen viste det sig at der ikke er implementeret understøttelse for at fjerne versioner fra et billede. Det betyder i praksisk at det ikke er muligt at slette en version hvis den først er blevet tilknyttet et billede, og at billede i det hele taget ikke kan slettes. Jævnfør eventuelt testkoden i billag D for nærmere beskrivelse af afprøvningen.

9 Konklusion

Målet for rapporten var at designe og implementeree en fungerende prototype af fotoarkivet, samt undersøge hvilken lagringsmetode, der var bedst egnet.

Ved rapportens afslutning må det konkluderes at rapporten ikke opfylder alle mål.

Følgende er blevet færdig beskrevet og implementeret.

Afprøvning af lagringsmetoder . Der blev afprøvet 3 forskellige metoder, og et filsystem viste sig at være mest optimal måde at lagre data på.

Design af datamodel Ud fra kravspecifikationen blev der udformet en datamodel, der kunne støtte et system, der levede op til kravspecifikationen.

Entitets framework . Der blev implementeret et framework hvori entiteter, deres relationer samt konsistensregler kunne implementeres.

Afprøvning De implementerede entiteter blev afprøvet og viste sig at fungere som forventet.

Det viste sig dog at der ikke var tid til at leve op til alle mål. Dette er i særdeleshed gået ud over implementeringen.

I skrivende stund mangler der følgende

Implementering af de sidste entiteter

Brugergrænsefladen der skal benytte det underliggende API.

Derudover viste det sig under afprøvningen af koden at der mangler at blive implementeret fjernelse af versioner fra billeder.

Det skal endvidere nævnes at der er blevet opdaget en uhensigtsmæssighed i site mappet. Der mangler muligheden for at oprette en kategori. Denne funktion eksisterer i API'et, men er ikke blevet beskrevet i brugervejledningen.

På trods af disse mangler mener vi dog at projektet og rapporten er en succes. Vi har implementeret, dokumenteret og afprøvet et framework samt nok tilhørende kode til, at vi mener, at projektet kan overtages og fortsættes af Det Danske Spejderkorps.

I forbinderlsen med testen blev vi af vejleder gjort opmærksom på artiklen [dat], der beskriver håndtering af filer i database sammenhæng på en anden måde. Denne kan anbefales som supplerende læsning.

Figurer

10 Kilder

10.1 Refererede

[dat] Version management and recoverability for large object data. `http://hssl.cs.jhu.edu/~randal/burns_mmdbms98.pdf`.

[Gra02] Mark Grand. *Patterns in Java*. WILEY, 2002.

[PEA] Pear - php extension and application repository. `http://pear.php.net/`.

[pgm] Postgresql 7.4.2 documentation. `http://www.postgresql.org/docs/7.4/static/`.

A Dansk/Engelsk oversættelse af termer brugt i koden

Dansk	Engelsk
kategori	category
nøgleord	keyword
billede	image
versioner	version
opgave	assignment
brugere	user/fotouser*
fil	file

* "user" er et reserveret ord i PostgreSQL, og tabellen, der indeholder brugere, er derfor kaldt fotouser i stedet for user.

B SQL-implementering af datamodellen

```
CREATE TABLE "assignment" (
    name character varying NOT NULL,
    deadline character varying NOT NULL,
    description character varying NOT NULL,
    id serial NOT NULL,
    coordinator integer NOT NULL
);

CREATE TABLE billedetest (
    name text,
    data bytea,
    mimetype character varying(100),
    id serial NOT NULL
) WITHOUT OIDS;

CREATE TABLE image (
    name character varying(100),
    description text,
    id serial NOT NULL,
    shutter character varying(10),
    aperture character varying(10),
    iso smallint,
    focal_length smallint,
    photographer_id integer,
    category_id integer
) WITHOUT OIDS;

CREATE TABLE keyword (
    word character varying(100) NOT NULL,
    id serial NOT NULL
) WITHOUT OIDS;


CREATE TABLE category (
    name character varying(100) NOT NULL,
    id serial NOT NULL,
    parent integer
) WITHOUT OIDS;

CREATE TABLE file (
    id serial NOT NULL,
    mimetype character varying(40) NOT NULL,
    "path" character varying(255) NOT NULL,
    name character varying(100)
) WITHOUT OIDS;

CREATE TABLE fotouser (
    id serial NOT NULL,
    login character varying(100) NOT NULL,
```

```
    "password" character varying(100) NOT NULL,
    permission integer DEFAULT 0 NOT NULL
) WITHOUT OIDS;

CREATE TABLE "version" (
    height smallint NOT NULL,
    width smallint NOT NULL,
    description text,
    id serial NOT NULL,
    image_id integer,
    file_id integer NOT NULL,
    name character varying(100)
) WITHOUT OIDS;

CREATE TABLE assignment_file (
    assignment_id integer NOT NULL,
    file_id integer NOT NULL
) WITHOUT OIDS;

CREATE TABLE assignment_image (
    assignment_id integer NOT NULL,
    image_id integer NOT NULL
) WITHOUT OIDS;

CREATE TABLE assigned_photographer (
    assignment_id integer NOT NULL,
    photographer_id integer NOT NULL
) WITHOUT OIDS;


CREATE TABLE image_keyword (
    image_id integer NOT NULL,
    keyword_id integer NOT NULL
) WITHOUT OIDS;

ALTER TABLE ONLY "assignment"
    ADD CONSTRAINT assignment_pkey PRIMARY KEY (id);

ALTER TABLE ONLY assigned_photographer
    ADD CONSTRAINT assigned_photographer_pkey
        PRIMARY KEY (assignment_id, photographer_id);

ALTER TABLE ONLY assignment_file
    ADD CONSTRAINT assignment_file_pkey
        PRIMARY KEY (assignment_id, file_id);

ALTER TABLE ONLY assignment_image
    ADD CONSTRAINT assignment_image_pkey
        PRIMARY KEY (image_id, assignment_id);

ALTER TABLE ONLY category
```

```
    ADD CONSTRAINT category_pkey PRIMARY KEY (id);

ALTER TABLE ONLY file
    ADD CONSTRAINT file_pkey PRIMARY KEY (id);

ALTER TABLE ONLY fotouser
    ADD CONSTRAINT fotouser_pkey PRIMARY KEY (id);

ALTER TABLE ONLY image
    ADD CONSTRAINT image_pkey PRIMARY KEY (id);

ALTER TABLE ONLY image_keyword
    ADD CONSTRAINT image_keyword_pkey
        PRIMARY KEY (keyword_id, image_id);

ALTER TABLE ONLY keyword
    ADD CONSTRAINT keyword_pkey PRIMARY KEY (id);

ALTER TABLE ONLY "version"
    ADD CONSTRAINT version_pkey PRIMARY KEY (id);

ALTER TABLE ONLY assignment_file
    ADD CONSTRAINT "$1" FOREIGN KEY (assignment_id)
        REFERENCES "assignment"(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY "assignment"
    ADD CONSTRAINT "$1" FOREIGN KEY (coordinator)
        REFERENCES fotouser(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY assignment_image
    ADD CONSTRAINT "$1" FOREIGN KEY (assignment_id)
        REFERENCES "assignment"(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY assigned_photographer
    ADD CONSTRAINT "$1" FOREIGN KEY (assignment_id)
        REFERENCES "assignment"(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY assigned_photographer
    ADD CONSTRAINT "$2" FOREIGN KEY (photographer_id)
        REFERENCES fotouser(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;
```

```
ALTER TABLE ONLY image_keyword
    ADD CONSTRAINT "$1" FOREIGN KEY (keyword_id)
        REFERENCES keyword(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY image_keyword
    ADD CONSTRAINT "$2" FOREIGN KEY (image_id)
        REFERENCES image(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY image
    ADD CONSTRAINT "$1" FOREIGN KEY (photographer_id)
        REFERENCES fotouser(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY image
    ADD CONSTRAINT "$2" FOREIGN KEY (photographer_id)
        REFERENCES category(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY "version"
    ADD CONSTRAINT "$1" FOREIGN KEY (file_id)
        REFERENCES file(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY "version"
    ADD CONSTRAINT "$2" FOREIGN KEY (image_id)
        REFERENCES image(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY assignment_file
    ADD CONSTRAINT "$3" FOREIGN KEY (file_id)
        REFERENCES file(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY assignment_image
    ADD CONSTRAINT "$3" FOREIGN KEY (image_id)
        REFERENCES image(id)
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;

ALTER TABLE ONLY category
    ADD CONSTRAINT "$1" FOREIGN KEY (parent)
        REFERENCES category(id)
```

```
            ON UPDATE RESTRICT
            ON DELETE RESTRICT;
```

C Test af lagringsmetoder for billeder

C.1 PHP-implementering af test1

PHP-implementeringen er delt op i en række filer med hver sin rolle. For hver lagringsmetode er der implementeret en klasse til indlæsning af billeder, og en stedfortræder der bruges til at udhente billedet igen. Når billedet skal udhentes sker det ved at gennemføre en HTTP-forspørgsel der henter billedet igennem stedfortræderen, og altså ikke ved at hente billedet direkte fra filsystem eller database.

runtest1.php Starter selve testen

test1.class.php Indeholder logikken der udføre de forskellige dele af testen

imagtest.class.php Indeholder generel kode der bruges af de enkelte implementeringer af de forskellige lagringsmetoder.

fstest.class.php Implementering af filsystems-lagring

blobtest.class.php Implementeringa af lagring i Large Objects

byteatest.class.php Implementering af lagring vha. bytea

blobimage.php Implementering af en stedfortræder der bruges til at udhente billeder lagret i Large Objects

byteaiage.php Implementering af en stedfortræder der bruges til at udhente billeder lagret vha. bytea

fsimage.php Implementering af en stedfortræder der bruges til at udhente billeder lagret i filsystemet

blobimage.php

Page 1/1

```php
<?php
include("common.php");

// list
$oid = $_REQUEST['id'];
$result = pg_query($conn, "SELECT mime FROM image_blob where data = $oid");

if(!$result || !pg_numrows($result)){
    die("Could not retrive image: " . pg_result_error($result));
}

$row = pg_fetch_row($result);
$mime = $row[0];

pg_exec($conn, "begin");

// get a handler
$loid = pg_loopen($conn, $oid, "r");
header("Content-type: $mime");
pg_lo_read_all($loid);

// http://lxr.php.net/source/php-src/ext/pgsql/pgsql.c
pg_loclose($loid);

//done
pg_query($conn, "commit");
?>
```

blobtest.class.php

Page 1/2

```php
<?php
include_once("imagetest.class.php");

class blobtest extends imagetest{
    var $conn;
    var $myname = "blob";


    function blobtest(){
        $this->dbConnect();
        $this->http_get = $this->webroot . "/blobimage.php?id=";
    }

    function clean(){
        $result = pg_query($this->conn, "SELECT name,data FROM image_blob");

        if(!$result){
            die("Could not retrive list: " . pg_result_error($result));
        }
        if($this->debug) print("Clearing " . pg_num_rows($result) . " old records \n");

        while($row = pg_fetch_row($result)){
            $name = $row[0];
            $oid = $row[1];
            if($this->debug) print("- " . $name . " ");
            pg_exec($this->conn, "begin");
            if($ures = pg_lo_unlink($this->conn, $oid)){
                if($this->debug) print("ok\n");
            }else{
                print("could note delete $name : " . pg_result_error($ures) . "\n");
            }
            pg_query($this->conn, "DELETE FROM image_blob WHERE data = " . $oid);
            pg_query($this->conn, "commit");
        }
    }

    function insertblob($path, $mime){
        $filename = basename($path);
        $size = filesize($path);
        $fp = fopen($path, "r");
        $buffer = fread($fp, filesize($path));
        fclose($fp);

        $time_start = $this->getmicrotime();
        pg_exec($this->conn, "begin");

        $oid = pg_locreate($this->conn);
        if($this->debug) print("- oid $oid\n");

        $rs = pg_exec($this->conn,"INSERT INTO image_blob(name, mime, size, data) VALUES('$filen
ame', '$mime', $size, $oid);");

        $handle = pg_loopen ($this->conn, $oid, "w");
        pg_lowrite ($handle, $buffer);
        pg_loclose ($handle);

        pg_exec($this->conn, "commit");
        $time_end = $this->getmicrotime();
        $time = $time_end - $time_start;

        // return whatever is needed to get the data back
        return($oid);
```

blobtest.class.php Page 2/2

```
    }

    function run($images){
        $this->clean();
        $ids = array();
        foreach($images as $path){
            $ids[] = $this->insert($path, "image/jpeg");
        }

        foreach($ids as $id){
            $this->httpget($id);
        }
    }
}
?>
```

byteaimage.php Page 1/1

```
<?php
include("common.php");

$id = $_REQUEST['id'];

$result = pg_query("SELECT mime,size, data from image_bytea WHERE id = $id");

if(!$result || !pg_numrows($result)){
    die("Could not retrive image: " . pg_result_error($result));
}
$row = pg_fetch_row($result);

header("Content-Type: " . $row[0]);
header("Content-Length: " . $row[1]);

echo pg_unescape_bytea($row[2]);
?>
```

byteatest.class.php

Page 1/2

```
<?php
include_once("imagetest.class.php");

class byteatest extends imagetest{

    var $conn;
    var $myname = "bytea";


    function byteatest(){
        $this->dbConnect();
        $this->http_get = $this->webroot . "/byteaimage.php?id=";
    }

    function clean(){
        $result = pg_query($this->conn, "SELECT id, name FROM image_bytea");

        if(!$result){
            die("Could not retrive list: " . pg_result_error($result));
        }
        if($this->debug) print("Clearing " . pg_num_rows($result) . " old records \n");

        while($row = pg_fetch_row($result)){
            $id = $row[0];
            $name = $row[1];
            if($this->debug) print("- " . $name . "\n");
            $res_del = pg_query($this->conn, "DELETE FROM image_bytea WHERE id = " . $id
);

            if(!$res_del){
                print(" Could not delete $name: " . pg_error_result($res_del) . "\n");
            }else{
                if($this->debug) print("ok \n");
            }
        }
    }

    function insertbytea($path, $mime){
        if($this->debug)  print("inserting $path\n");
        $filename = basename($path);

        $size = filesize($path);
        $fp = fopen($path, "r");
        $buffer = fread($fp, filesize($path));
        $time1 = $this->getmicrotime();
        $data = pg_escape_bytea($buffer);
        print("escapetime: " . ($this->getmicrotime() - $time1) . "\n");

        fclose($fp);

        pg_exec($this->conn, "begin");

        // get the id
        $res_id=pg_query($this->conn, "SELECT nextval('image_bytea_id_seq') as key");
        $id = pg_fetch_array($res_id, 0);
        $id=$id['key'];

        if($this->debug) print("- id $id\n");

        $rs = pg_exec($this->conn,"INSERT INTO image_bytea(id, name, mime, size, data) VALUES('$i
d', '$mime', $size, $id, '$data'::bytea)");
        pg_exec($this->conn, "commit");

        return $id;
```

byteatest.class.php

Page 2/2

```
    }

    function run($images){
        $this->clean();
        $ids = array();
        foreach($images as $path){
            $ids[] = $this->insert($path, "image/jpeg");
        }

        foreach($ids as $id){
            $this->httpget($id);
        }
    }
}
?>
```

fsimage.php

Page 1/1

```php
<?php
include("common.php");

$storage_dir = "storage";

$id = $_REQUEST['id'];

$result = pg_query("SELECT mime, path, size from image_fs WHERE id = $id");

if(!$result || !pg_numrows($result)){
    die("Could not retrive image: " . pg_result_error($result));
}
$row = pg_fetch_row($result);

$mime = $row[0];
$path = $row[1];
$size = $row[2];

$image_path = $storage_dir ."/". $path;
if(!is_file($image_path)){
    die("error: $image_path is not a file");
}

header("Content-Type: $mime");
header("Content-Length: " . $size);

readfile($image_path);
?>
```

fstest.class.php

Page 1/2

```php
<?php
include_once("imagetest.class.php");

class fstest extends imagetest{

    var $conn;
    var $myname = "fs";

    var $dir = "storage";

    function fstest(){
        $this->dbConnect();
        $this->http_get = $this->webroot . "/fsimage.php?id=";
    }

    function clean(){
        $result = pg_query($this->conn, "SELECT id, path FROM image_fs");

        if(!$result){
            die("Could not retrive list: " . pg_result_error($result));
        }
        if($this->debug) print("Clearing " . pg_num_rows($result) . " old records \n");

        while($row = pg_fetch_row($result)){
            $id = $row[0];
            $path = $row[1];
            if($this->debug) print("- " . $path . " ");

            if(unlink($this->dir . "/" . $path)){

                $res_del = pg_query($this->conn, "DELETE FROM image_fs WHERE id = $id");
                if(!$res_del){
                    print("failed (".pg_result_error($res_del).")\n");
                }else{
                    if($this->debug)  print("ok\n");
                }

            }else{
                print("unlink of " . $this->dir . "/" . $path . " failed\n");
            }
        }
    }

    function insertfs($path, $mime){
        if($this->debug) print("inserting $path\n");
        $filename = basename($path);
        $size = filesize($path);

        // get the id
        $res_id=pg_query($this->conn, "SELECT nextval('image_fs_id_seq') as key");
        $id = pg_fetch_array($res_id, 0);
        $id = $id['key'];
        if($this->debug) print("- id $id\n");

        $pathinfo = pathinfo($filename);
        $storage_path = $id . ($pathinfo['extension'] ? "." . $pathinfo['extension'] : "");
        copy($path, $this->dir . "/" . $storage_path);

        $rs = pg_exec($this->conn,"INSERT INTO image_fs(id, name, mime, size, path) VALUES($id, '
$name', '$mime', $size, '$storage_path')");

        return $id;
    }
```

fstest.class.php

Page 2/2

```
    function run($images){
        $this->clean();
        $ids = array();
        foreach($images as $path){
            $ids[] = $this->insert($path, "image/jpeg");
        }

        foreach($ids as $id){
            $this->httpget($id);
        }
    }
}
?>
```

imagetest.class.php

Page 1/2

```
<?php

class imagetest{
    var $conn;
    var $webroot = "http://127.0.0.1/foto/imagetest/dbvsfs";
    var $myname = "undef";
    var $debug = false;


    function dbConnect(){
        $this->conn = pg_connect("host='127.0.0.1' dbname='fotoarkiv' user='fotoarkiv' password=''");

    }

    function getmicrotime(){
        list($usec, $sec) = explode(" ",microtime());
        return ((float)$usec + (float)$sec);
    }

    function httpget($id){

        $size = 0;
        $url = $this->http_get . "$id";

        $time_start = $this->getmicrotime();
        $fp = fopen($url, "r");
        // do first read

        $data = fread($fp, 8192);
        $time_first = $this->getmicrotime();
        case {
            if (strlen($data) == 0) {
                break;
            } else{
                $size += strlen($data);
            }
            $data = fread($fp, 8192);
        } while(true);
        fclose($fp);

        $time_end = $this->getmicrotime();
        $time = $time_end - $time_start;
        $time2 = $time_first - $time_start;

        print($this->myname . ":read:$id:$size:$time\n");
        print($this->myname . ":readfrist:$id:$size:$time2\n");
    }

    function insertundef($junk1, $junk2){
        die("undefined insert");
    }

    function insert($path, $mime){
        $function = insert . $this->myname;

        $time_start = $this->getmicrotime();

        $id = $this->$function($path,$mime);

        $time_end = $this->getmicrotime();
        $time = $time_end - $time_start;

        $size = filesize($path);
```

imagetest.class.php Page 2/2

```
        print($this->myname. ":insert:$id:$size:$time\n");

        return $id;
    }
}

?>
```

runtest1.php Page 1/1

```
<?php
include("test1.class.php");
$test1 = new test1();
$test1->clean();
sleep(10);
$test1->run();
?>
```

test1.class.php


```
<?php
include("blobtest.class.php");
include("byteatest.class.php");
include("fstest.class.php");

class test1{
    var $blob;
    var $bytea;
    var $fs;

    var $images = array();

    function test1(){

        // Images of different sizes
        $images = array("200.jpg", "500.jpg", "1000.jpg", "2000.jpg", "4000.jpg", "8000.tif",
 "12000.tif", "16000.tif", "20000.tif", "24000.tif", "28000.tif", "32000.tif");

        foreach($images as $id => $image){
            $this->images[$id] = "/mirror/fotoarkiv_test/sizes/" . $image;
        }

        $this->blob = new blobtest();
        $this->bytea = new byteatest();
        $this->fs = new fstest();
    }

    function clean(){
        $this->blob->clean();
        $this->bytea->clean();
        $this->fs->clean();
    }

    function run(){
        $methods = array($this->blob, $this->bytea, $this->fs);

        foreach($methods as $method){
            $ids = array();
            foreach($this->images as $image){
                // allow the system to settle
                sleep(3);
                $pathinfo = pathinfo($image);

                print("test1:" . $pathinfo['basename'] . ":");
                $imagemap[$pathinfo['basename']] = $method->insert($image, "image/jpeg"
);
            }
            foreach($imagemap as $imagename => $id){
                // allow the system to settle
                sleep(3);
                print("test1:" . $imagename . ":");
                $method->httpget($id);
            }
        }
    }
}
?>
```

C.2 SQL-beskrivelse af tabeller

Large Object

Struktur af tabellen der blev brugt ved test af lagring af billeder med Large Objects:

```
CREATE TABLE image_blob (
    name character varying(100),
    mime character varying(100),
    data oid,
    size integer
) WITHOUT OIDS;
```

Data-attributten indeholder et id der referere til den interne tabel `pg_largeobject` hvor alle Large Object bliver lagret. Det er ikke muligt at definere referencer til system-kataloger. Strukturen af tabellen er som følger:

```
CREATE TABLE pg_largeobject (
    loid oid NOT NULL,
    pageno integer NOT NULL,
    data bytea
) WITHOUT OIDS;
```

Bytea

Struktur af tabellen der blev brugt ved test af lagring af billeder med bytea:

```
CREATE TABLE image_bytea (
    id serial NOT NULL,
    name character varying(100),
    mime character varying(100),
    size integer,
    data bytea
) WITHOUT OIDS;
```

Filsystem

Struktur af tabellen der blev brugt ved test af lagring af billeder i et filsystem:

```
CREATE TABLE image_fs (
    id serial NOT NULL,
    name character varying(100),
    mime character varying(100),
    size integer,
    "path" character varying(255)
) WITHOUT OIDS;
```

Path-attributten er en relativ sti til filen i filsystemet.

D Afprøvningsresultater

D.1 Testoutput

```
TestCase category->testcreate() passed
TestCase category->testset() passed
TestCase category->testdeletewithparent() passed
TestCase category->testget() passed
TestCase category->testdeletewithimage() failed:
            delete image expected true, actual false
TestCase category->testdeletewithimage() failed:
            delete without image expected true, actual false

TestCase image->testcreate() passed
TestCase image->testset() passed
TestCase image->testprotection() passed
TestCase image->testdelete() failed:
            remove versions expected true, actual false
TestCase image->testdelete() failed:
            delete image without versions expected true, actual false
TestCase image->testget() failed:
            get version expected true, actual false

TestCase file->testcreate() passed
TestCase file->testset() passed
TestCase file->testdelete() passed
TestCase file->testget() passed

TestCase version->testcreate() passed
TestCase version->testset() passed
TestCase version->testdelete() passed
TestCase version->testget() passed
TestCase version->testprotection() passed
```

D.2 Kildekode

Category.test.php

Page 1/2

```
<?php
require_once 'PHPUnit.php';
/**
* Test of File
* @package test
*/
class Category extends PHPUnit_TestCase
{
    function Category($name) {
        $this->PHPUnit_TestCase($name);
    }

    function testCreate(){
        $category = new CategoryDAO();
        $this->assertTrue($category->set('name', "testcategory'"));
        $this->assertTrue($category->create());
    }

    function testSet(){
        $category = new CategoryDAO();

        $parent = new CategoryDAO();
        $this->assertTrue($parent->set('name', "parent"), "set parent");
        $this->assertTrue($parent->create(), "Store parent");

        $this->assertTrue($category->set("parent", $parent), "set parent");
        $this->assertFalse($category->create(), "Store without name");
    }

    // return a version
    function getVersion(){
        $file = new FileDAO();
        $file->set('name', 'test.jpg');
        $file->set('data', 'd:/test.jpg');
        $file->create();

        $version = new VersionDAO();
        $version->set('file', $file);
        $version->set('name', "version1");
        $version->create();

    }

    function testDeleteWithParent(){
        $category = new CategoryDAO();
        $category->set("name", "testcategory");

        $parent = new CategoryDAO();
        $parent->set("name", "parent");
        $parent->create();

        $category->set("parent", $parent);
        $category->create();



        $this->assertFalse($parent->delete(), "delete parent with child");
        $this->assertTrue($category->delete(), "delete child");
        $this->assertTrue($parent->delete(), "delete parent without child");
    }
```

Category.test.php

Page 2/2

```
    function testDeleteWithImage(){
        $category = new CategoryDAO();
        $category->set("name", "testcategory");

        $image = new ImageDAO();
        $image->set('description',"test description");
        $image->set('name','testname');
        $image->set('category',$category);
        $image->create();

        $this->assertFalse($category->delete(), "delete with image");
        $this->assertTrue($image->delete(), "delete image");
        $this->assertTrue($category->delete(), "delete without image");
    }

    function testGet(){
        $category = new CategoryDAO();

        $parent = new CategoryDAO();
        $parent->set('name', "testparent");
        $parent->create();

        $category->set("parent", $parent);
        $category->set("name", "testname");
        $id = $category->create();

        $category1 = new CategoryDAO();
        $category1->load($id);

        $this->assertEquals($category->get('name'), "testname");
        $this->assertTrue($parent = $category->get('parent'));
        $this->assertEquals($parent->get('name'), "testparent");
    }
}

?>
```

File.test.php

Page 1/2

```
<?php
require_once 'PHPUnit.php';
/**
* Test of File
* @package test
*/
class File extends PHPUnit_TestCase
{
    function File($name) {
        $this->PHPUnit_TestCase($name);
    }

    function testCreate(){
        $file = new FileDAO();
        $this->assertTrue($file->set('name', 'test.jpg'));
        $this->assertTrue($file->set('data', 'd:/test.jpg'));
        $this->assertTrue($file->create());
    }

    function testSet(){
        $file = new FileDAO();
        $this->assertFalse($file->create(), "create empty");

        $this->assertTrue($file->set('name', 'test.jpg'), "set name");
        $this->assertFalse($file->create());

        $file = new FileDAO();
        $this->assertTrue($file->set('data', 'd:/test.jpg'));
        $this->assertFalse($file->create());
    }

    function testDelete(){
        $file = new FileDAO();
        $this->assertTrue($file->set('name', 'test.jpg'), "set name");
        $this->assertTrue($file->set('data', 'd:/test.jpg'), "set data");
        $this->assertTrue($file->create(), "create file");

        $version = new VersionDAO();
        $this->assertTrue($version->set('file', $file), "set file");
        $this->assertTrue($version->set('name', "version1"), "set name");
        $this->assertTrue($version->create(), "create version");

        $this->assertFalse($file->delete(), "Delete file with version");
        $this->assertTrue($version->delete(), "Delete version");
        $this->assertTrue($file->delete(), "Delete file without version");

    }

    function testGet(){
        $file = new FileDAO();
        $this->assertTrue($file->set('name', 'testname'), "set name");
        $this->assertTrue($file->set('data', 'd:/test.jpg'), "set data");
        $this->assertTrue($id = $file->create(), "store file");

        $file1 = new FileDAO();
        $this->assertTrue($file->load($id), "retrive file again");
        $this->assertTrue($file->get('name') == 'testname', "test for name");
    }

}
```

File.test.php

Page 2/2

```
?>
```

Image.test.php Page 1/3

```
<?php
require_once 'PHPUnit.php';
/**
* Test of Image
* @package test
*/
class Image extends PHPUnit_TestCase
{
    function Image($name) {
        $this->PHPUnit_TestCase($name);
    }



    //helper
    // helper
        function getCategory(){
        $category = new CategoryDAO();
        $category->set('name', "tempcat");
        $category->create();

        return $category;
    }

    //helper
    function getVersion(){
        $file = new FileDAO();
        $file->set('name', 'tempfile');
        $file->set("data", "d:/test.jpg");
        $file->create();

        $version = new VersionDAO();
        $version->set('file', $file);
        $version->set('name', "tempversion");
        $version->create();

        return $version;
    }



      function testCreate(){
        $image = new ImageDAO();

        $this->assertTrue($image->set('description',"1337 description"), "set description");
        $this->assertTrue($image->set('shutter','1/125'), "set shutter");
        $this->assertTrue($image->set('aperture','5.6f'), "set aperture");
        $this->assertTrue($image->set('name','testname'), "set name");
        $this->assertTrue($image->set('iso',800), "set iso");
        $this->assertTrue($image->set('focal_length',22), "set focal");
        $this->assertTrue($image->set('category',$this->getCategory()), "set category");
        $this->assertTrue($image->create(), "creation");
        $this->assertTrue($image->addVersion($this->getVersion()), "set version");
        $this->assertTrue($image->store(), "creation");

    }

    function testSet(){
        $category = $this->getCategory();
```

Image.test.php Page 2/3

```
        $image = new ImageDAO();
        $this->assertTrue($image->set('name','testname'));
        $this->assertTrue($image->set('category',$category));
        $this->assertFalse($image->create(), "Creation missing description");


        $image = new ImageDAO();
        $this->assertTrue($image->set('description',"1337 description"));
        $this->assertTrue($image->set('category',$category));
        $this->assertFalse($image->create(), "Creation missing name");


        $image = new ImageDAO();
        $this->assertTrue($image->set('description',"1337 description"));
        $this->assertTrue($image->set('name','testname'));
        $this->assertFalse($image->create(), "Creation missing category");

    }

    function testDelete(){
        $image = new ImageDAO();

        $this->assertTrue($image->set('description',"1337 description"));
        $this->assertTrue($image->set('name','testname'));
        $this->assertTrue($image->set('focal_length',22));
        $this->assertTrue($image->set('category',$this->getCategory()));
        $this->assertTrue($image->create(), "creation");
        $this->assertTrue($image->addVersion($this->getVersion()));
        $this->assertTrue($image->store(), "update");

        $this->assertFalse($image->delete(), "delete image with versions");
        $this->assertTrue($image->removeAllVersions(), "remove versions");
        $this->assertTrue($image->delete(), "delete image without versions");
    }

    function testGet(){
        $image = new ImageDAO();
        $category = $this->getCategory();
        $version = $this->getVersion();


        $this->assertTrue($image->set('description',"test description"), "set description");
        $this->assertTrue($image->set('shutter','tests'), "set shutter");
        $this->assertTrue($image->set('aperture','testa'), "set aperture");
        $this->assertTrue($image->set('name','testname1'), "set name");
        $this->assertTrue($image->set('iso',999), "set iso");
        $this->assertTrue($image->set('focal_length',99), "set focal");
        $this->assertTrue($image->set('category',$category), "set category");
        $this->assertTrue($id = $image->create(), "creation");
        $this->assertTrue($image->addVersion($version), "add version");
        $this->assertTrue($id = $image->store(), "update");


        $image1 = new ImageDAO();
        $this->assertTrue($image1->load($id), "load image");

        $this->assertEquals($image1->get('description'),"test description", "get description");
        $this->assertEquals($image1->get('shutter'),"tests", "get shutter");
        $this->assertEquals($image1->get('aperture'),"testa", "get aperture");
        $this->assertEquals($image1->get('name'),"testname1", "get name");
        $this->assertEquals($image1->get('iso'),999, "get iso");
```

Image.test.php

Page 3/3

```
        $this->assertEquals($image1->get('focal_length'),99, "get focal");

        $this->assertTrue($category1 = $image1->get('category'), "retrive category");
        if(is_a($category1, "categorydao")){
            $this->assertEquals($category1->get('name'), $category->get('name'), "chec
k category");
        }else{
            $this->fail("Did not get a category back");
        }


        $this->assertTrue($version1 = $image1->getVersions(), "get version");
    }

    function testProtection(){
    }
}

?>
```

RunTests.php

Page 1/1

```
<?php
require_once 'PHPUnit.php';
require_once("../setup.inc.php");
define("DEBUG", false);

// list of our tests, remember to include the file as well
$tests = array("Category", "Image", "File", "Version");

// run each test
foreach($tests as $test){
    require($test . ".test.php");
    $suite  = new PHPUnit_TestSuite($test);
    $result = PHPUnit::run($suite);

    echo $result -> toHTML();
    if(DEBUG){
    print("<br><hr>");
    print(Core::debugAsHTML());
    Core::flushDebugContents();
    }
}

?>
```

Version.test.php Page 1/2

```
<?php
require_once 'PHPUnit.php';
/**
* Test of File
* @package test
*/
class Version extends PHPUnit_TestCase
{
    function Version($name) {
        $this->PHPUnit_TestCase($name);
    }

    // helper
    function createFile(){
        $file = new FileDAO();
        $file->set('name', 'test.jpg');
        $file->set("data", "d:/test.jpg");
        $file->create();
        return $file;
    }

    function testCreate(){
        $file = $this->createFile();

        $version = new VersionDAO();
        $this->assertTrue($version->set('file', $file), "assign file");
        $this->assertTrue($version->set('name', "version1"), "set name");
        $this->assertTrue($version->create());
    }

    function testSet(){
        $file = $this->createFile();

        $version = new VersionDAO();
        $this->assertTrue($version->set('file', $file), "assign file");
        $this->assertFalse($version->create());

        $version = new VersionDAO();
        $this->assertTrue($version->set('name', "version1"), "set name");
        $this->assertFalse($version->create());
    }

    function testDelete(){
        $file = $this->createFile();

        $version = new VersionDAO();
        $this->assertTrue($version->set('file', $file), "assign file");
        $this->assertTrue($version->set('name', "version1"), "set name");
        $this->assertTrue($version->create());

        // associate with image
        // try to delete
    }

    function testGet(){
        $file = $this->createFile();
        $this->assertTrue($file->get('name') == "test.jpg");

        $version = new VersionDAO();
        $this->assertTrue($version->set('file', $file), "assign file");
        $this->assertTrue($version->set('name', "nametest"), "set name");
```

Version.test.php Page 2/2

```
        $this->assertTrue($id = $version->create(), "Create version");


        $version1 = new VersionDAO();
        $this->assertTrue($version1->load($id), "Try to load");
        $this->assertEquals($version1->get('name'), "nametest", "recheck name");
        $this->assertTrue($file = $version1->get('file'), "retrive file");
        $this->assertTrue($file->get('name') == "test.jpg", "test filename");
    }

    function testProtection(){
        // file_id and image_id must be changed manually
        $version = new VersionDAO();
        $this->assertFalse($version->set('file_id', 1), "Try to set file_id");
        $this->assertFalse($version->set('image_id', 1), "Try to set image_id");
    }

}

?>
```

E API-dokumentation

Dokumentationen er genereret af phpDocumentor der kan hentes på `http://www.phpdoc.org/`.

Fotoarkiv API-dokumentation

Contents

Package DAO Procedural Elements

CategoryDAO.class.php

- **Package** DAO
- **Filesource** Source Code for this file

DAO.class.php

- **Package** DAO
- **Filesource** Source Code for this file

FileDAO.class.php

- **Package** DAO
- **Filesource** Source Code for this file

ImageDAO.class.php

- **Package** DAO
- **Filesource** Source Code for this file

VersionDAO.class.php

- **Package** DAO
- **Filesource** Source Code for this file

Package DAO Classes

Class CategoryDAO

[line 6]

Represents a Catagory entity

- **Package** DAO

CategoryDAO::$data

mixed = array(
"id" => null,
"name" => null,
"parent" => null
) *[line 9]*

CategoryDAO::$data_id

mixed = "id" *[line 17]*

CategoryDAO::$data_table

mixed = "category" *[line 16]*

CategoryDAO::$db

mixed = *[line 19]*

CategoryDAO::$myName

mixed = __CLASS__ *[line 15]*

CategoryDAO::$parent

mixed = null *[line 22]*

function CategoryDAO::get($key) *[line 86]*
function CategoryDAO::isDisposable() *[line 113]*
function CategoryDAO::isStorable() *[line 103]*
function CategoryDAO::postLoad() *[line 24]*
function CategoryDAO::preCreate() *[line 48]*
function CategoryDAO::preStore() *[line 41]*
function CategoryDAO::set($key, $value) *[line 62]*

Class DAO

[line 9]

Data Access Object

Parent class for all Data Access Object. The purpose of the DAO is to encapsulate A number of entities while allowing for data-constraints to be maintained

- **Package** DAO

DAO::$data

array = array() *[line 39]*

The list of attributes in the entity

- **Access** protected
- **Abstract Element**

DAO::$data_id

string = null *[line 55]*

Primary key identifying the entity instance

- **Access** protected
- **Abstract Element**

DAO::$data_table

string = null *[line 47]*

Name of the table storing the entity data

- **Access** protected
- **Abstract Element**

DAO::$db

DB = *[line 31]*

A PEAR DB instance, intialized by the DAO constructor

- **Access** protected

DAO::$loaded

boolean = false *[line 16]*

Tells wether the object has been loaded from DB yet

- **Access** public

Constructor function DAO::DAO([$id = null]) *[line 62]*

Constructor, will be called as long as the subclasses dont implement one

- **Access** protected

int|boolean function DAO::create() *[line 302]*

Generic create function.

This function must be called instead of store() to store the first instance of a entity

- **Final**
- **Access** public

boolean function DAO::delete() *[line 274]*

Deletes an instance of the entity. The function disposable() is consultet before the entity is deleted

- **Final**
- **Access** public

mixed function DAO::get($key) *[line 337]*

Function Parameters:

- *string* **$key**

Generic getter

Returns the value of an entity-attribute. This function could be overridden by a subclass in order to provide a proxy for a value.

- **Access** public

boolean function DAO::isDisposable() *[line 183]*

Indicates whether the instance can be deleted

This will most likely be a check of wether other entity instances are referreing to this instance. *Must be overloaded by subclasses*, the default implementation *will* * fail

- **Abstract Element**
- **Access** protected

boolean function DAO::isStorable() *[line 199]*

Indicates whether the instance can be stored

This will most likely be a check of the required attributes and relations to other entity instanses. *Must be overloaded by subclasses*, the default implementation *will* fail

- **Abstract Element**
- **Access** protected

boolean function DAO::load($id) *[line 95]*

Function Parameters:

- *int* **$id** ID of the entity instance to load

Loads an instance with data

This will load the files sepecified in $data from the table specified by $table After load, the postLoad() method will be invoked.

- **Final**
- **Access** public

boolean function DAO::preCreate() *[line 169]*

Initialize any data needed before creation.

Mainly for fetching a new ID from a sequence. This function should be overloaded by any subclasses of DAO if they require preperation before creation.

- **Access** protected

boolean function DAO::preStore() *[line 155]*

Prepares the entity instance for storage

Local objects should be correted to their primitive value, associated dataobjects should * be stored, and so on. Should be overloaded by subclass if it needs to prepare before storage.

- **Abstract Element**
- **Access** protected

boolean function DAO::set($key, $value) *[line 323]*

Function Parameters:

- *string* **$key** name of the attribute to set
- *mixed* **$value** the valuse to set

Generic setter

As this function will modify the state of the object it must be overloaded in order

to guarantee consistency. If the function is not overloaded it will cause the application to halt !

- **Access** public

boolean function DAO::store() *[line 216]*

Generic store-function.

This function will store the status of the entity. This is mainly done by storing the values of $data into $data_table. The function will first consult preStore() in order to ensure that the Entity is in a valid state.

- **Final**
- **Access** public

Class FileDAO

[line 11]

Represents an instance of the File entity.

File have the following attributes:

- id
- mimtype, of the file
- path, to the file

- **Package** DAO

FileDAO::$data

mixed = array(
"id" => null,
"name" => null,
"mimetype" => null,
"path" => null
) *[line 13]*

FileDAO::$data_id

mixed = "id" *[line 22]*

FileDAO::$data_table

mixed = "file" *[line 21]*

FileDAO::$db

mixed = *[line 25]*

FileDAO::$myName

mixed = __CLASS__ *[line 20]*

FileDAO::$newfile

mixed = *[line 28]*

function FileDAO::get($key) *[line 101]*
function FileDAO::isDisposable() *[line 135]*
function FileDAO::isStorable() *[line 114]*
function FileDAO::preCreate() *[line 30]*
function FileDAO::set($key, $value) *[line 66]*
function FileDAO::storeFile($file) *[line 51]*

Class ImageDAO

[line 6]

Represents an Image Entity instanse

- **Package** DAO

ImageDAO::$category

mixed = null *[line 27]*

ImageDAO::$data

mixed = array(
"id" => null,
"name" => null,
"description" => null,
"shutter" => null,
"aperture" => null,
"iso" => NULL,
"focal_length" => null,
"photographer_id" => null,
"category_id" => null
) *[line 9]*

ImageDAO::$data_id

mixed = "id" *[line 23]*

ImageDAO::$data_table

mixed = "image" *[line 22]*

ImageDAO::$db

mixed = *[line 25]*

ImageDAO::$myName

mixed = __CLASS__ *[line 21]*

ImageDAO::$versions

mixed = array() *[line 28]*

function ImageDAO::addVersion($version) *[line 120]*
function ImageDAO::get($key) *[line 103]*
function ImageDAO::getVersions() *[line 139]*
function ImageDAO::isDisposable() *[line 50]*
function ImageDAO::isStorable() *[line 59]*
function ImageDAO::postLoad() *[line 30]*
function ImageDAO::preCreate() *[line 67]*
function ImageDAO::preStore() *[line 44]*
function ImageDAO::removeAllVersions() *[line 130]*
function ImageDAO::set($key, $value) *[line 79]*

Class VersionDAO

[line 6]

Represents an Version Entity instanse

- **Package** DAO

VersionDAO::$data

mixed = array(
"id" => null,
"description" => null,
"height" => 0, // just for now until we can extract the info from the file
"width" => 0, // just for now until we can extract the info from the file
"name" => null,
"image_id" => null,
"file_id" => null
) *[line 9]*

VersionDAO::$data_id

mixed = "id" *[line 21]*

VersionDAO::$data_table

mixed = "version" *[line 20]*

VersionDAO::$db

mixed = *[line 24]*

VersionDAO::$file

mixed = *[line 26]*

VersionDAO::$image

mixed = *[line 27]*

VersionDAO::$myName

mixed = __CLASS__ *[line 19]*

function VersionDAO::get($key) *[line 68]*
array function VersionDAO::getVersionsOf($image_id) *[line 143]*
Function Parameters:

- *int* **$image_id**

Returns a list of VersionDAOs representing versions of an image

- **Static**
- **Access** public

function VersionDAO::isDisposable() *[line 59]*
function VersionDAO::isStorable() *[line 53]*
function VersionDAO::postLoad() *[line 30]*
function VersionDAO::preCreate() *[line 41]*
function VersionDAO::preStore() *[line 36]*
function VersionDAO::set($key, $value) *[line 89]*

Appendices

Appendix A - Class Trees

Package DAO

DAO

- DAO
  - CategoryDAO
  - FileDAO
  - ImageDAO
  - VersionDAO

Appendix C - Source Code

Package DAO

File Source for CategoryDAO.class.php

Documentation for this file is available at [CategoryDAO.class.php]

```
1     <?php
2     /**
3     * Represents a Catagory entity
4     * @package    DAO
5     */
6     class CategoryDAO extends DAO{
7
8         // Settings specific for this instances
8
9         var $data = array(
10        "id"            =>    null,
11        "name"            =>    null,
12        "parent"            =>    null
13        );
14
15        var $myName =  __CLASS__;
16        var $data_table = "category"          ;
17        var $data_id =  "id"          ;
18
19        var $db;
20
21        // extra
21
22        var $parent = null;
23
24        function postLoad(){
25            // setup parent
26            if($this->   data['parent'] != null &&         $this->   data['parent']){
27                $parent = new CategoryDAO($this->   data['parent']);
28            }else{
29                return true;
30            }
31
32            if($parent){
33                $this->   parent = $parent;
34                return true;
35            }else{
36                Core::err($this->   myName . "->postLoad"             , "     Could not instansiate
parent $parent"     );
37                return false;
38            }
39        }
40
41        function preStore(){
42            if($this->   parent !== null){
43                $this->   data['parent'] = $this->   parent->   get('id');
44            }
45            return true;
46        }
47
48        function preCreate(){
49            // fetch new id
50            $id = $this->   db->   nextId('category_id');
51
52            if(DB::isError($id)){
53                Core::err($this->   myName . "->preCreate"             , "Could not fetch next
ID:"      . Core::getDbErr($id));
54                return false;
55            }
56            $this->   data['id'] = $id;
57            return true;
58        }
59
60
61
62          function set($key, $value){
63            switch($key){
```

```
64                 case "parent"          :
65                 if(is_a($value, "categoryDAO"          ) &&         $value->   loaded){
66                     $this->   parent = $value;
67                     $this->   data['parent'] = $value->   get('id');
68                     return true;
69                 }else{
70                     return false;
71                 }
72                 break;
73
74                 default:
75                 if(array_key_exists($key, $this->   data)){
76                     $this->   data[$key] = $value;
77                     return true;
78                 }else{
79                     Core::crit($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
80                     return false;
81                 }
82                 break;
83             }
84         }
85
86         function get($key){
87             switch($key){
88                 case "parent"          :
89                 return $this->   parent;
90                 break;
91
92                 default:
93                 if(array_key_exists($key, $this->   data)){
94                     return $this->   data[$key];
95                 }else{
96                     Core::err($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
97                     return false;
98                 }
99                 break;
100            }
101        }
102
103         function isStorable(){
104             if($this->   data['name'] == null){
105                 Core::debug($this->   myName . "->isStorable"             , "Missing name, cannot
store"     );
106                 return false;
107             }else{
108                 return true;
109             }
110        }
111
112
113        function isDisposable(){
114            // check that we'r not referred to by any other category
115            $db = $this->   db;
116
117            $sth = $db->   prepare("SELECT id FROM category WHERE parent = ?"          );
118            $result = $db->   execute($sth, array($this->   data['id']));
119            if(DB::isError($result)){
120                Core::crit($this->   myName . "->isDisposable"             , "Could not retrive
list: "      . Core::getDbErr($result));
121
122                return false;
123            }
124
125            if($result->   numRows() >    0){
126                $row = $result->   fetchRow();
127                Core::err($this->   myName . "->isDisposable"             , "This instance is
still referred to by Category instance instance "      . $row['id']);
128                return false;
129            }
130
131            // check that we'r not referred to by any other image
132            $sth = $db->   prepare("SELECT id FROM image WHERE category_id = ?"          );
133            $result = $db->   execute($sth, array($this->   data['id']));
134            if(DB::isError($result)){
135                Core::crit($this->   myName . "->isDisposable"             , "Could not retrive
list: "      . Core::getDbErr($result));
136                return false;
137            }
```

```
138
139             if($result->   numRows() >    0){
140                 $row = $result->   fetchRow();
141                 Core::err($this->   myName . "->isDisposable"             , "This instance is
still referred to by Image instance instance "      . $row['id']);
142                 return false;
143             }
144             return true;
145         }
146     }
147     ?>
```

File Source for DAO.class.php

Documentation for this file is available at [DAO.class.php]

```
1     <?php
2     /**
3     * Data Access Object
4     *
5     * Parent class for all Data Access Object. The purpose of the DAO is to encapsulate
6     * A number of entities while allowing for data-constraints to be maintained
7     * @package DAO
8     */
9     class DAO{
10
11        /**
12        * Tells wether the object has been loaded from DB yet
13        * @var boolean
14        * @access public
15        */
16        var $loaded = false;
17
18        /**
19        * Tells wether the current data exists in the database or should be insertet
20        * when stored
21        * @var boolean
22        * @access private
23        */
24        var $fresh = false;
25
26        /**
27        * A PEAR DB instance, intialized by the DAO constructor
28        * @var DB
29        * @access protected
30        */
31        var $db;
32
33        /**
34        * The list of attributes in the entity
35        * @abstract
36        * @var array
37        * @access protected
38        */
39        var $data = array();
40
41        /**
42        * Name of the table storing the entity data
43        * @abstract
44        * @var string
45        * @access protected
46        */
47        var $data_table = null;
48
49        /**
50        * Primary key identifying the entity instance
51        * @abstract
52        * @var string
53        * @access protected
54        */
55        var $data_id = null;
56
57        /**
58        * Constructor, will be called as long as the subclasses dont implement one
59        *
60        * @access protected
61        */
62        function DAO($id = null){
63            $this->   connect();
64            Core::debug($this->   myName . "->constructor"             , "Creating new
object"      );
65            if($id !== null &&         is_numeric($id)){
66                Core::debug($this->   myName . "->constructor"             , "     loading with id
```

```
$id"     );
                $this->   load($id);
            }
        }

        /**
        * Connects to database, called automatically from constructor
        *
        * @access    private
        * @final
        * @return    void
        */
        function connect(){
            // forbind til databasen
            $this->   db = Core::getDB();
        }


        /**
        * Loads an instance with data
        *
        * This will load the files sepecified in $data from the table specified by $table
        * After load, the postLoad() method will be invoked.
        *
        * @param int $id ID of the entity instance to load
        * @access public
        * @final
        * @return boolean indication of the load-status
        */
        function load($id){
            Core::debug($this->   myName . "->load"             , "Loading object "           . $id);

            $db = $this->   db;

            // try to fetch the data
            $sth = $db->   prepare("SELECT "           . implode(", "           , array_keys($this-
>   data)) ." FROM "           . $this->   data_table . " WHERE "           . $this->   data_id . "
= ?"     );
            $result = $db->   execute($sth, array($id));

            if(DB::isError($result)){
                Core::crit("Database error, could not retrive row: "           .
Core::getDbErr($result));
                return false;
            }

            $row = $result->   fetchRow();

            if($row){
                $this->   data = $row;

                // let the inheritor get a chance to update the loaded data, create obejcts and so on

                if($this->   postLoad()){
                    $this->   loaded = true;
                    Core::debug($this->   myName . "->load"             , "Object loaded"          );
                    return true;
                }else{
                    Core::err($this->   myName . "->load"             , "Postload failed"          );
                    return false;
                }
            }else{
                Core::err($this->   myname . " ->load"             , "Could not load object with
id "      . $id);
                return false;
            }
        }

        /**
        * Implements any extra initialization needed after a load
        *
        * Should be overloaded if the subclass needs special initialization, instantion of objects
        * and so on. Should not be used to enforce constraints as we assume the a newly loaded
        * entity will be valid. See isStorable
        *
        * @access   private
        * @return   boolean the entity instance will not be allowed to load unless postLoad returns true
        */
        function postLoad(){
            return true;
```

```
142         }
143
144
145         /**
146         * Prepares the entity instance for storage
147         *
148         * Local objects should be correted to their primitive value, associated dataobjects should * be
stored, and so on. Should be overloaded by subclass if it needs to prepare before
149         * storage.
150         *
151         * @access    protected
152         * @abstract
153         * @return    boolean the object will not be allowed to be stored unless preStore returns true
154         */
155         function preStore(){
156             return true;
157         }
158
159
160         /**
161         * Initialize any data needed before creation.
162         *
163         * Mainly for fetching a new ID from a sequence. This function should be overloaded by any
164         * subclasses of DAO if they require preperation before creation.
165         *
166         * @access   protected
167         * @return   boolean the object will not be created unless preCreate returns true
168         */
169         function preCreate(){
170             return true;
171         }
172
173         /**
174         * Indicates whether the instance can be deleted
175         *
176         * This will most likely be a check of wether other entity instances are referreing to this
177         * instance. <i>Must be overloaded by subclasses</i>, the default implementation
<i>will</i> * fail
178         *
179         * @access   protected
180         * @abstract
181         * @return   boolean the object will not be created unless preCreate returns true
182         */
183         function isDisposable(){
184             Core::crit($this->   myName . "->isDisposable()"             , "Should be
overloaded!"     );
185             return false;
186         }
187
188         /**
189         * Indicates whether the instance can be stored
190         *
191         * This will most likely be a check of the required attributes and relations to other entity
192         * instanses. <i>Must be overloaded by subclasses</i>, the default implementation
193         * <i>will</i> fail
194         *
195         * @access   protected
196         * @abstract
197         * @return   boolean the object will not be created unless preCreate returns true
198         */
199         function isStorable(){
200             Core::crit($this->   myName . "->isStorable()"             , "Should be
overloaded!"     );
201             return false;
202
203         }
204
205         /**
206         * Generic store-function.
207         *
208         * This function will store the status of the entity. This is mainly done by storing the
209         * values of $data into $data_table. The function will first consult preStore() in order to
210         * ensure that the Entity is in a valid state.
211         *
212         * @access   public
213         * @final
214         * @return    boolean
215         */
216         function store(){
217             Core::debug($this->   myName . "->store"             , "Attempting to store object
```

```
"     );
            $db = $this->   db;

            if(!$this->   fresh &&         $this->   loaded){
                if($this->   isStorable()){
                    // update the data
                    $result = $db->   autoExecute($this->   data_table, $this->   data,
                    DB_AUTOQUERY_UPDATE, $this->   data_id . ' = ' . $this->   data[$this->   data_id]);

                    if(DB::isError($result)){
                        Core::err($this->   myName . "->store"             , "Database error,
could not store object: "      . Core::getDbErr($result));
                        return false;
                    }else{
                        //done
                        Core::debug($this->   myName . "->store"             , "Objected
stored"     );
                        return $this->   data[$this->   data_id];
                    }
                }else{
                    Core::err($this->   myName . "->store"             , "The object is not
storable in its current state"     );
                }
            }else if($this->   fresh){
                // create a new tuple
                if($this->   preCreate()){
                    if($this->   isStorable()){
                        $result = $db->   autoExecute($this->   data_table, $this->   data,
DB_AUTOQUERY_INSERT);

                        if(DB::isError($result)){
                            Core::err($this->   myName . "->store"             , "Database error,
could not create object: "      . Core::getDbErr($result));
                            return false;
                        }else{
                            //done
                            Core::debug($this->   myName . "->store"             , "Objected
created"     );
                            $this->   loaded = true;
                            return $this->   data[$this->   data_id];
                        }
                    }else{
                        Core::err($this->   myName . "->store"             , "The object is not
storable in its current state"     );
                    }
                }else{
                    Core::err($this->   myName . "->store"             , "Could not prepare for
creation"     );
                    return false;
                }
            }else{
                // must not happen
                Core::crit($this->   myName . "->store"             , "Attempt to store unloaded
non-fresh object !"     );
                return false;
            }
        }

        /**
        * Deletes an instance of the entity. The function disposable() is consultet before the
        * entity is deleted
        *
        * @access    public
        * @final
        * @return    boolean
        */
        function delete(){
            Core::debug($this->   myName . "->delete"             , "Attempting to delete object
"      . $this->   data[$this->   data_id]);

            if($this->   isDisposable() &&         $this->   loaded){
                $result = $this->   db->   query("DELETE FROM "           . $this->   data_table .
" WHERE "           . $this->   data_id . " = "           . $this->   data[$this->   data_id]);

                if(DB::isError($result)){
                    Core::err($this->   myName . "->delete()"             , "Could not delete
object !"     );
                    return false;
                }else{
                    return true;
```

```
285                 }
286
287             }else{
288                 Core::err($this->   myName . "->delete"             , "The object cannot be
deleted in its present state !"     );
289                 return false;
290             }
291         }
292
293         /**
294         * Generic create function.
295         *
296         * This function must be called instead of store() to store the first instance of a entity
297         *
298         * @access    public
299         * @final
300         * @return    int|booleaneither the id of the new entity or false
301         */
302         function create(){
303             if($this->   loaded){
304                 Core::crit($this->   myName . "->create"             , "Attemt to create loaded
DAO"     );
305             }
306
307             $this->   fresh = true;
308             return $this->   store();
309         }
310
311         /**
312         * Generic setter
313         *
314         * <i> As this function will modify the state of the object it must be overloaded in order
315         * to guarantee consistency.</i>
316         * If the function is not overloaded it will cause the application to halt !
317         *
318         * @param    string $key name of the attribute to set
319         * @param    mixed $value the valuse to set
320         * @access    public
321         * @return    boolean
322         */
323         function set($key, $value){
324             die("set-function should be overloaded"          );
325         }
326
327         /**
328         * Generic getter
329         *
330         * Returns the value of an entity-attribute. This function could be overridden by a subclass
331         * in order to provide a proxy for a value.
332         *
333         * @param    string $key
334         * @access    public
335         * @return    mixed value of the attribute $key
336         */
337         function get($key){
338             switch($key){
339                 default:
340                 if(array_key_exists($key, $this->   data)){
341                     return $this->   data[$key];
342                 }else{
343                     Core::crit($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
344                     return false;
345                 }
346                 break;
347             }
348         }
349     }
350
351     ?>
```

File Source for FileDAO.class.php

Documentation for this file is available at [FileDAO.class.php]

```
1     <?php
2     /**
3     * Represents an instance of the File entity.
4     *
5     * File have the following attributes:
6     * - id
7     * - mimtype, of the file
8     * - path, to the file
9     * @package  DAO
10    */
11    class FileDAO extends DAO{
12
13        var $data = array(
14        "id"            =>    null,
15        "name"            =>    null,
16        "mimetype"            =>    null,
17        "path"            =>    null
18        );
19
20        var $myName =  __CLASS__;
21        var $data_table = "file"          ;
22        var $data_id =  "id"          ;
23
24        // database connection, initalized by super-class
24
25        var $db;
26
27        // set if file is updated
27
28        var $newfile;
29
30        function preCreate(){
31            if($this->   newfile){
32                // fetch new id
33                $id = $this->   db->   nextId('file_id');
34
35                if(DB::isError($id)){
36                    Core::err($this->   myName . "->preCreate"             , "Could not fetch next
ID:"      . Core::getDbErr($id));
37                    return false;
38                }
39                $this->   data['id'] = $id;
40
41                if(! $this->   storeFile($this->   newfile)){
42                    return false;
43                }
44                return true;
45            }else{
46                Core::debug("FileDAO->preCreate"             , "Tried to create new object without
a file"     );
47                return false;
48            }
49        }
50
51        function storeFile($file){
52            $pathinfo = pathinfo($this->   newfile);
53            $path = $this->   data['id'] . "."            . $pathinfo['extension'];
54            $abspath = FILE_STORAGE . "/"           . $path;
55            if(copy($this->   newfile, $abspath)){
56
57                Core::debug("FileDAO->set"             , "     coping $value to "      . $abspath);
58                $this->   data['path'] = $path;
59                $this->   data['mimetype'] = mime_content_type($abspath);
60                return true;
61            }else{
62                return false;
63            }
```

```
64          }
65
66          function set($key, $value){
67              switch($key){
68                  case "path"          :
69                      Core::err($this->   myName . "->set()"             , "Attempt to set
path"     );
70                      return false;
71                  break;
72
73                  case "data"          :
74                  if(is_file($value)){
75                      // delay until we know our id
76                      if(!$this->   loaded){
77                          $this->   newfile = $value;
78                          return true;
79                      }else{
80                          // else go ahead and try
81                          return $this->   store($value);
82                      }
83                  }else{
84                      Core::err($this->   myName . "->set()"             , "Attempt set data to
something that is not a file: "       . $value);
85                      return false;
86                  }
87                  break;
88
89                  default:
90                  if(array_key_exists($key, $this->   data)){
91                      $this->   data[$key] = $value;
92                      return true;
93                  }else{
94                      Core::crit($this->   myName . "->set()"             , "     Nonexistent field
'$key'"     );
95                      return false;
96                  }
97                  break;
98              }
99          }
100
101         function get($key){
102             switch($key){
103                 default:
104                 if(array_key_exists($key, $this->   data)){
105                     return $this->   data[$key];
106                 }else{
107                     Core::err($this->   myName . "->set()"             , "     Nonexistent field
'$key'"     );
108                     return false;
109                 }
110                 break;
111             }
112         }
113
114         function isStorable(){
115             if($this->   data['name'] == null){
116                 Core::err($this->   myName . "->isStorable"             , "Name is null"          );
117                 return false;
118             }
119
120             if($this->   data['path'] == null){
121                 Core::err($this->   myName . "->isStorable"             , "Path is null"          );
122                 return false;
123             }
124
125             // check that the file exists
126             if(!is_file(FILE_STORAGE . "/"           . $this->   data['path'])){
127                 Core::err($this->   myName . "->isStorable"             , "Could not find
file:"      . FILE_STORAGE . "/"           . $this->   data['path']);
128                 return false;
129             }
130
131             // all is good
132             return true;
133         }
134
135         function isDisposable(){
136             $db = $this->   db;
137
138             $sth = $db->   prepare("SELECT id FROM version WHERE file_id = ?"          );
```

```
139         $result = $db->   execute($sth, array($this->   data['id']));
140
141         if(DB::isError($result)){
142             Core::crit($this->   myName . "->isDisposable"             , "Database error:
"      . Core::getDbErr($result));
143             return false;
144         }
145
146         if($result->   numRows() >    0){
147             $row = $result->   fetchRow();
148
149             Core::err($this->   myName . "->isDisposable"             , "This instance is
still referred to by Version instance "      . $row['id']);
150             return false;
151         }
152
153         return true;
154     }
155 }
156 ?>
```

File Source for ImageDAO.class.php

Documentation for this file is available at [ImageDAO.class.php]

```
1       <?php
2       /**
3       * Represents an Image Entity instanse
4       * @package    DAO
5       */
6       class ImageDAO extends DAO{
7
8           // Settings specific for this instances
8
9           var $data = array(
10          "id"            =>    null,
11          "name"            =>    null,
12          "description"            =>    null,
13          "shutter"            =>    null,
14          "aperture"            =>    null,
15          "iso"            =>    NULL,
16          "focal_length"            =>    null,
17          "photographer_id"            =>    null,
18          "category_id"            =>    null
19          );
20
21          var $myName =  __CLASS__;
22          var $data_table = "image"           ;
23          var $data_id =  "id"           ;
24
25          var $db;
26
27          var $category = null;
28          var $versions = array();
29
30          function postLoad(){
31
32              // load category
33              $category = new CategoryDAO();
34              if(!$category->   load($this->   data['category_id'])){
35                  Core::err("Could not load my category"           );
36                  return false;
37              }else{
38                  $this->   category = $category;
39                  return true;
40              }
41          }
42
43
44          function preStore(){
45              $this->   data['category_id'] = $this->   category->   get('id');
46              return true;
47          }
48
49
50          function isDisposable(){
51              if(count($this->   versions) == 0){
52                  return true;
53              }else{
54                  Core::err($this->   myName . "->isDisposable"              , "Cannot be deleted as
long as versions exsts, use removeAllVerions() to clear"      );
55                  return false;
56              }
57          }
58
59          function isStorable(){
60              if($this->   data['name'] != null &&         $this->   data['description'] != null
&&         $this->   category != null){
61                  return true;
62              }else{
63                  return false;
64              }
```

```
65          }
66
67          function preCreate(){
68              // fetch new id
69              $id = $this->   db->   nextId('image_id');
70
71              if(DB::isError($id)){
72                  Core::err($this->   myName . "->preCreate"             , "Could not fetch next
ID:"      . Core::getDbErr($id));
73                  return false;
74              }
75              $this->   data['id'] = $id;
76              return true;
77          }
78
79          function set($key, $value){
80              switch($key){
81                  case "category"          :
82                  if(is_a($value, "categoryDAO"          ) &&         $value->   loaded){
83                      $this->   category = $value;
84                      $this->   data['category_id'] = $value->   get('id');
85                      return true;
86                  }else{
87                      return false;
88                  }
89                  break;
90
91                  default:
92                  if(array_key_exists($key, $this->   data)){
93                      $this->   data[$key] = $value;
94                      return true;
95                  }else{
96                      Core::crit($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
97                      return false;
98                  }
99                  break;
100             }
101         }
102
103         function get($key){
104             switch($key){
105                 case "category"          :
106                     return $this->   category;
107                 break;
108
109                 default:
110                 if(array_key_exists($key, $this->   data)){
111                     return $this->   data[$key];
112                 }else{
113                     Core::err($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
114                     return false;
115                 }
116                 break;
117             }
118         }
119
120         function addVersion($version){
121             if($version->   set('image', $this)){
122                 $this->   versions[] = $version;
123                 return $version->   store();
124             }else{
125                 Core::err($this->   myName . "->addVersion"             , "Could not update
Version"     );
126                 return false;
127             }
128         }
129
130         function removeAllVersions(){
131                 // get all versions
132                 // get their files
133                 // delete the files
134                 // delete the versions
135               Core::err($this->   myName . "->removeAlleVersions"             , "Missing !"          );
136               return false;
137         }
138
139         function getVersions(){
140             return  VersionDAO::getVersionsOf($this->   data['id']);
```

```
141         }
142
143     }
144     ?>
```

File Source for VersionDAO.class.php

Documentation for this file is available at [VersionDAO.class.php]

```
1     <?php
2     /**
3     * Represents an Version Entity instanse
4     * @package    DAO
5     */
6     class VersionDAO extends DAO{
7
8         // Settings specific for this instances
8
9         var $data = array(
10        "id"           =>    null,
11        "description"           =>    null,
12        "height"           =>    0, // just for now until we can extract the info from the file
12
13        "width"           =>    0, // just for now until we can extract the info from the file
13
14        "name"           =>    null,
15        "image_id"           =>    null,
16        "file_id"           =>    null
17        );
18
19        var $myName =  __CLASS__;
20        var $data_table = "version"          ;
21        var $data_id =  "id"          ;
22
23        // database connection, initalized by super-class
23
24        var $db;
25
26        var $file;
27        var $image;
28
29
30        function postLoad(){
31            $this->   file = new FileDAO($this->   data['file_id']);
32            $this->   image = new ImageDAO($this->   data['image_id']);
33            return true;
34        }
35
36        function preStore(){
37            // extract id from file and image and put it into attributes
38            return true;
39        }
40
41        function preCreate(){
42            // fetch new id
43            $id = $this->   db->   nextId('version_id');
44
45            if(DB::isError($id)){
46                Core::err($this->   myName . "->preCreate"             , "Could not fetch next
ID:"      . Core::getDbErr($id));
47                return false;
48            }
49            $this->   data['id'] = $id;
50            return true;
51        }
52
53        function isStorable(){
54            // note that we'r allowed not to be associated with a Image
55            return ($this->   data['id'] != null &&         $this->   data['file_id'] &&         $this-
>   data['name'] != null);
56        }
57
58
59        function isDisposable(){
60            if($this->   data['image_id'] != null){
61                Core::err($this->   myName . "->isDisposable()"             , "Must be deleted via
```

```
the removeVersion function associated Image "      . $this->   data['image_id']);
                return false;
            }else{
                return true;
            }
        }

         function get($key){
            switch($key){
                case "image"          :
                    return $this->   image;
                break;

                case "file"          :
                    return $this->   file;
                break;

                default:
                if(array_key_exists($key, $this->   data)){
                    return $this->   data[$key];
                }else{
                    Core::crit($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
                    return false;
                }
                break;
            }
        }

          function set($key, $value){
            switch($key){
                case "file"          :
                if($value->   loaded){
                    $this->   data['file_id'] = $value->   get('id');
                    return true;
                }else{
                    Core::err($this->   myName . "->set()"             , "attempt to assign me an
unloaded File"     );
                    return false;
                }
                break;

                case "file_id"          :
                    Core::err($this->   myName . "->set()"             , "file_id cannot be
modified manually, use \"file\""               );
                    return false;
                break;

                case "image"          :
                if($value->   loaded){
                    $this->   data['image_id'] = $value->   get('id');
                    return true;
                }else{
                    Core::err($this->   myName . "->set()"             , "attempt to assign me an
unloaded Image"     );
                }
                break;

                case "image_id"          :
                    Core::err($this->   myName . "->set()"             , "image_id cannot be
modified manually, use \"image\""               );
                    return false;
                break;


                default:
                if(array_key_exists($key, $this->   data)){
                    $this->   data[$key] = $value;
                    return true;
                }else{
                    Core::crit($this->   myName . "->set()"             , "     Nonexistent field
$key"     );
                    return false;
                }
                break;
            }
        }
```

```
135         /**
136         * Returns a list of VersionDAOs representing versions of an image
137         *
138         * @param int $image_id
139         * @access public
140         * @return array of VersionDAO
141         * @static
142         */
143         function getVersionsOf($image_id){
144             $db = Core::getDB();
145
146             $sth = $db->   prepare("SELECT id FROM version WHERE image_id = ?"          );
147             $result = $db->   execute($sth, array($image_id));
148
149             $versions = array();
150
151             while($row = $result->   fetchRow()){
152                 $versions[] = new VersionDAO($row['id']);
153             }
154
155             Core::debug($this->   myName . "->getVersionsOf"             , "got "           .
count($versions) . " versions  for image "           . $image_id);
156             return $versions;
157         }
158     }
159     ?>
```

Index

C

D

F

I

V