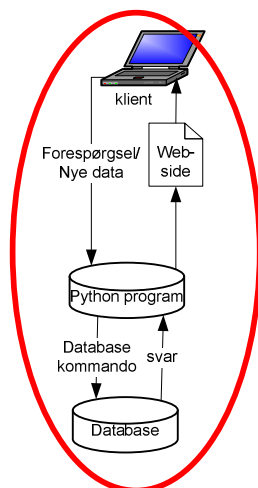


## Øvelse 10. Dynamiske web-sider og cgi-modulet

I denne opgave skal vi bygge de foregående opgaver sammen til en mini-udgave af det samlede system I skal aflevere til vintereksamen.



I får udleveret et lille system som I skal placere på jeres webpages-server. Systemet benytter sig af den database I lavede i øvelse 7.1 samt af det modul I færdiggjorde i øvelse 9. I kan se hvordan systemet kører på denne adresse:

<http://pba.web.student.hum.au.dk/opgave10/Frontpage.htm>

Selve systemet kan downloades fra kursussiden under navnet opgave10. Det er en zip-file. Pak zipfilen ud, læg modulerne i en mappe på jeres webpages, og få det til at køre.

1. Når I lægger filerne over på jeres web-pages, skal de ligge som ascii-filer. Derfor skal I bruge *Filezilla*-programmet til at flytte dem. I *Filezilla* sætter I *filetransfer* til at være *ascii* ved under *settings* → *ASCII/Binary*, at indføre *py* som suffix og sætte den på *autodetect*. Så lægger *Filezilla* dem korrekt over.
2. Husk at give *execute*-tilladelser. I *Filezilla* højreklikker I på den overførte file, så vælger I *fileattributes* og sætter kryds i *execute*.

I kender nogle af filerne i forvejen. Det gælder *database* som I arbejdede med i øvelse 6 og *OOoogsql* som I arbejdede med i øvelse 9. *htmlRoutines* indeholder to funktioner som laver html-tabeller. Den ene arbejdede I med i øvelse 5.

Øvelsen går ud på at læse *dynamicWebPages*, *searchResult* og *findResult* og finde ud af hvordan de virker, samt at gennemskue hvordan de forskellige moduler virker sammen. Meningen er at I skal tilpasse *opgave10* til det projekt I afleverer til vintereksamen.

Her er en kortfattet overordnet beskrivelse af systemet.

### *Opbygningen af systemet*

*dynamicWebPages* indeholder klasser der repræsenterer de dynamiske websider der er i systemet. Fig. 1 viser et klassediagram over disse web-sider. Alle sider er færdi-

ge undtagen én. Det er siden *actionResult* der skal genereres af et python-program. I stedet går linket til siden *actionResult.py* der blot fremviser indholdet af variabelen *fieldstorage*, der indeholder de data som python modtager.

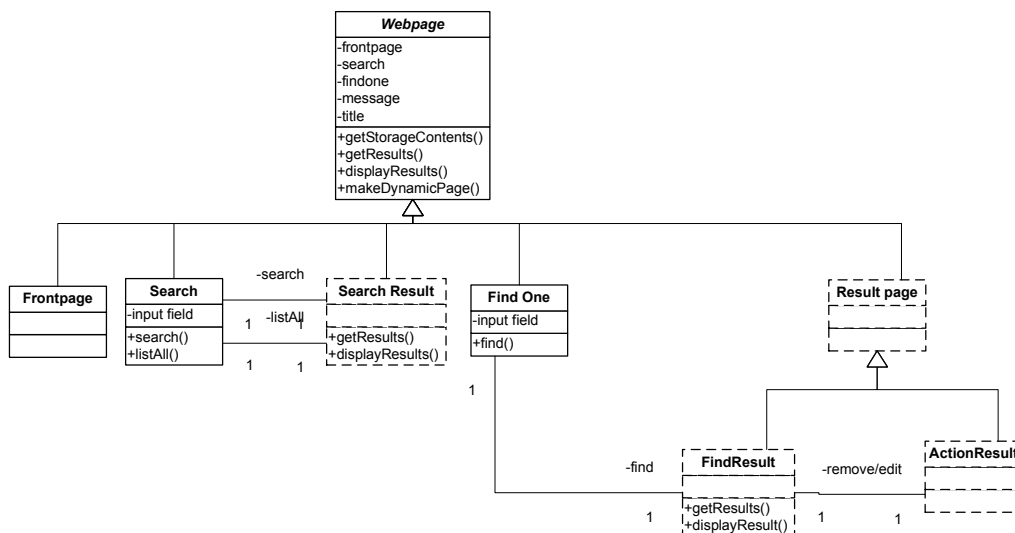
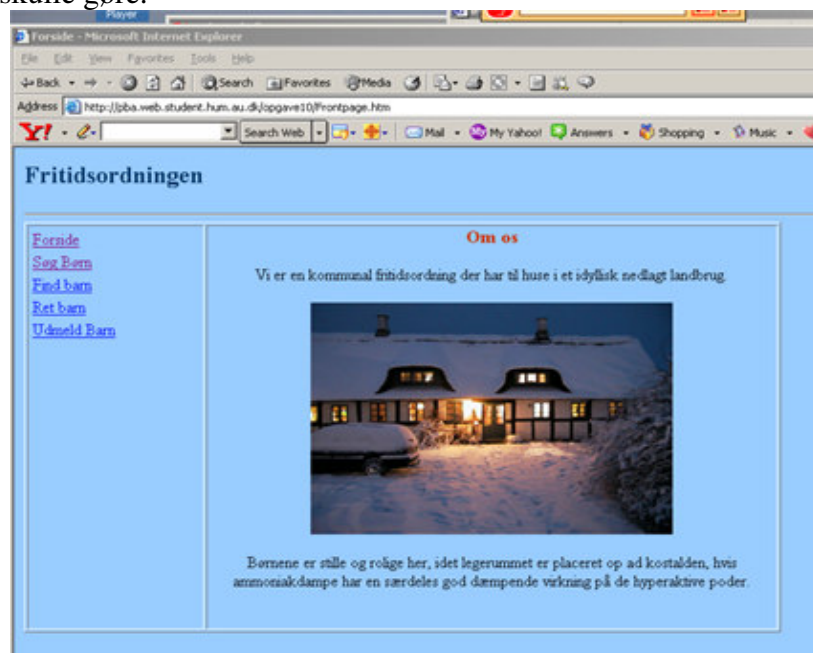


Fig. 1. Klassesdiagram

I diagrammet har jeg brugt følgende den konvention at almindelige linkadresser repræsenteres som egenskaber, mens knapper der er indlejret i en *form*, repræsenteres som metoder. Vi ser at alle klasser har links til *frontpage*, *search* og *findone*. Det er de links I kan se til venstre i skærbilledet. De går igen på alle siderne. *Find barn*, *Ret barn* og *Udmeld barn* linker alle til *FindOne*. Det er fordi vi først må fiske barnet frem før vi kan rette eller udmelde det, men hvis vi kun havde et link med navnet *Find Barn* i menuen, så ville en person, der ønsker at udmelde barnet, ikke vide hvad han skulle gøre.



2. Skærbillede

Web-siderne er et mix af håndlavede (fuldt optrukne i diagrammet) og dynamisk genererede (stiplede) sider. *Frontpage*, *search* og *findone* er håndlavede, mens *searchresult*, *findresult* og *actionresult* genereres af Python-programmer. Reglen jer har fulgt er, at alle sider, der modtager information fra web-klienten og fremfinder ting fra databasen, er dynamiske.

Når man vil sende information til Python sker det ved felter og knapper der findes indeni et *form*-tag. Submit-knapperne *Ret* og *Udmeld* samt den efterfølgende tekst-box er indlejret i et *form*-tag.

Fig. 3. Hvordan en 'form' ser ud på skærbilledet

Når man trykker på *udmeld*, viser næste webside hvad *fieldstorage* indeholder. Den har attributten 'Remove' hvis værdi er 'udmeld' samt attributten 'cpr' hvis værdi er '1111111111'. Python kan nu nemt fiske den oversendte information frem. Hvis vi har lagt vores *FieldStorage* i variabelen *storageContents* (`storageContents = cgi.FieldStorage()`), så kan vi fiske indholdet frem således:

```
Cpr = storageContents['cpr'].value
```

Vi opnåede at *FieldStorage* fik dette indhold med følgende html-kode:

```
<form name="form1" method="post" action="actionResult.py">
  <p>
    <input type="submit" name="Edit" value="Ret">
    <input type="submit" name="Remove" value="Udmeld">
    <input type="text" name="cpr" value = 1111111111>
  </p>
</form>
```

Vi ser at *Value* er det navn, elementet fremvises under på web-siden, eller indholdet af en textbox, mens *Name* er det navn data sendes til python under. Navnet på det python-program, der skal kaldes når vi trykker på en knap, specificeres i det første *form*-tag. I eksemplet ovenfor er det *actionResult.py*:

```
<form name="form1" method="post" action="actionResult.py">
```

I diagrammet i fig. 1 har jeg brugt en association til at angive hvilke python-programmer, en submit-knap aktiverer.

De python-programmer der kaldes via cgi skal opfylde følgende:

1. Programmet skal starte med

```
#!/usr/bin/python
```

Det fortæller serveren hvor den skal finde python-fortolkeren.

2. Alle moduler skal importere cgi modulet:

```
import cgi
```

3. Hvis man vil have python til at skrive fejlmeddelelser tilbage skal man skrive

```
import cgi, os, cgitb
cgitb.enable()
```

Ellers får man bare en intetsigende side tilbage i hovedet.

Jeg har forsøgt at konstruere modulerne så hvert modul kun har ét emne og kun ét ansvarsområde:

*findResult* og *searchResult* frembringer svar på en søgning. Den første forudsætter at der højst er ét svar, den sidste kan frembringe flere svar.

*DynamicWebpages* repræsenterer alle de dynamiske sider i systemet som klasser. Klasserne indeholder metoder for at få fat i data den modtager fra en klient, til at bede sql-modulet om at lave databasekald og til at fremvise sig selv som en web-side.

*findResult*-programmet laver en instans af klassen *FindResult*, og *searchResult* af klassen *SearchResult*. Dernæst printer de klassernes selv-fremvisning:

```
test = SearchResult(beginning, ending, message, title)
print test.makeDynamicPage()
```

Den printede html-kode sendes ikke til nogen printer eller konsol, men returneres til serveren, der sender den tilbage til klienten. Parametrene *beginning*, *ending*, *message*, og *title* indeholder følgende:

**Beginning:** den html-kode der er fælles for starten af alle siderne.

**Ending:** den html-kode der er fælles for slutningen af alle siderne.

**Message:** overskriften på højre side af skærmbilledet der varierer fra side til side

**Title:** titlen som web-siden der varierer fra side til side.

*Beginning* og *Ending* er globale variable, der udfyldes af modulet *dynamicWebPages*.

Som vist i fig. 4, er modulerne niveaudelt: de øvre moduler bruger de nedre, men ikke omvendt. Dette anses for godt design, fordi man så let kan tilføje moduler i den øvre ende uden at skulle rette på de nedre moduler. F.eks. er det nemt at tilføje flere web-sider på toppen.

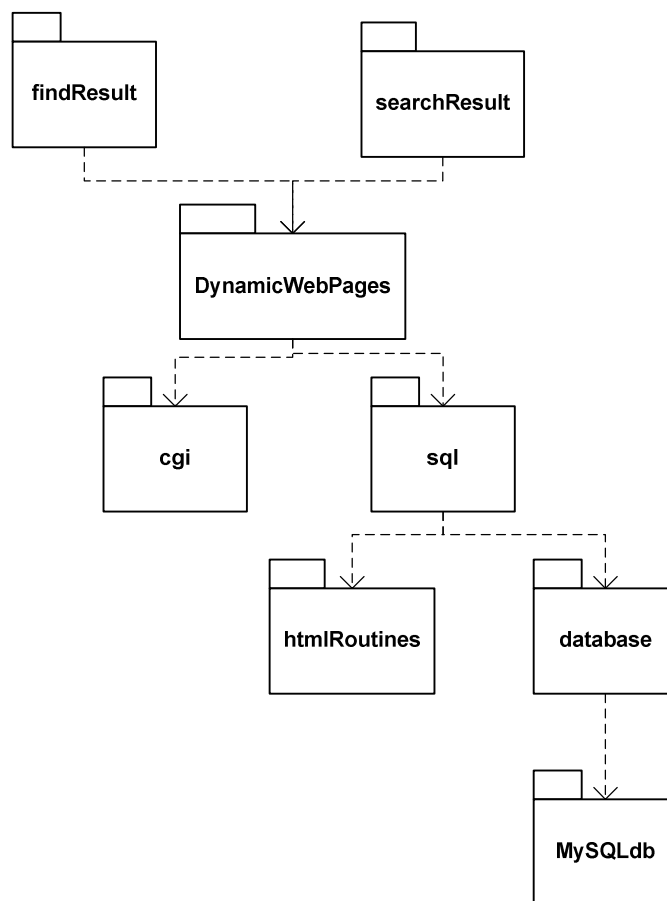


Fig. 4. Modulerne

Modulet *sql* bruges f.eks. af *dynamicWebPages*. F.eks. skal klassen *FindResult* returnere den række i databasen 'Child', der har et bestemt cpr-nummer. Det gør den ved at generere et objekt af typen *sql.Row* med cpr som parameter for en primærnøgle:

```
self.results = sql.Row('Child',self.cpr)
```

*dynamicWebPages* beskæftiger sig altså ikke med databasekald og den slags slibigheder. Det har den folk til. Dens ansvarsområde ligger udelukkende i at lave dynamiske sider, og databasekald har den *delegeret* til modulet *sql*.

*HTMLroutines* er specialiseret i at omdanne Pytons listestrukturer til html-tabeller. Disse metoder kaldes af *sql*-modulets klasser for at de kan visualisere sig selv som html-tabeller.

### *Design og fejlretning af dynamiske hjemmesider*

Husk allerførst, at alle sider der sendes tilbage til klienten skal starte med følgende:

```
Content-type: text/html
```

efterfulgt af et linjeskift. Dette fortæller klienten hvad der er for et dokument den modtager.

Når man laver et system, gælder det om at fejl-rette modul for modul. Man sørger for at de nedre moduler fungerer før man lader de øvre moduler bruge dem. Det kan man gøre efter konstruktionen

```
if __name__ == "__main__":
```

Den kode der står efter linjen bliver udført når programmet køres selvstændigt, men ikke når det køres som modul der importeres af et andet program. Man kan se et eksempel fra sql-modulet:

```
if __name__ == "__main__":
    print 'make a table-object and print it'
    aTable = Table('Child')
    aTable.printContents()
    print
    print 'fetch a row from Child with primary key 1111111111'
    aRow = Row('Child', '1111111111')
    print 'change the first name to Jeppe and store it'
    aRow.put('firstname', 'Jeppe')
    aRow.update()
```

Her testes Table og Row før *dynamicWebPages* får lov til at bruge disse klasser.

Når man udformer dynamiske hjemmesider, er der to gode tricks:

1. Lav først siden i hånden i DreamWeaver. Her kan I kæle for design og layout uden at kunne en linje html. Selektér dernæst i det nederste vindue den del som afhænger af modtaget information og databasekald. Kopier den selekterede html-koden ud af det øverste vindue og kig på den. Det er denne kode I skal få et Python-program til at generere. Kopier dernæst den foregående html-kode og læg den ind i en variabel (f.eks. *beginning* som gjort ovenfor). Kopier den efterfølgende kode og læg den ind i en variable (f.eks. *ending* som gjort ovenfor).

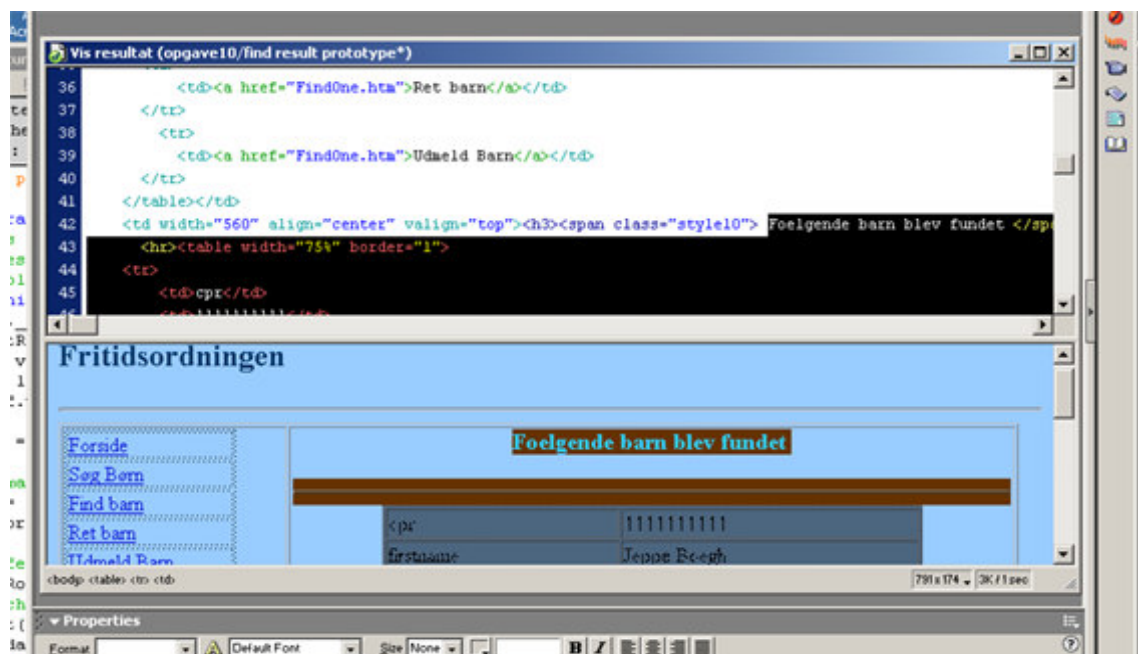


Fig. 5. Opmærkning i DreamWeaver.

Hvis funktionen `getResults()` laver den html-del der fremviser resultater, kan selve siden, *thePage*, fremstilles ved følgende to linjer:

```
thePage = '''Content-type: text/html
...
thePage += self.beginning + self.getResults()+ self.ending
```

2. Hvis I ikke lige har styr på hvad det egentlig er for ting der modtages gennem `FieldStorage`, kan I starte med at lade action-delen kalde siden *testresults.py*. Denne side gør ikke andet end at udskrive indholdet af *FieldStorage*. Hvis vi f.eks. lader siden *Search* kalde *testresults.py* får vi følgende svar:



Fig. 6. Fremvisning af `FieldStorage`.

Den viser øverst hvilke attributter og værdier `FieldStorage` indeholder. Der er f.eks. et attribut 'Fornavn' med værdien 'peter'. Nederst kan man se et rå udprint af `FieldStorage`.