Danair
Flysæde
Booking

Kim Rotbøl-Larsen
020273-1201

Indhold

Problemformulering **4**
Baggrunds beskrivelse 4
Problemformulering 4
Afgrænsninger 6
Metode 7
Disposition 7
Foranalyse **8**
Udvælgelse af entiteter 8
ER-Diagrammet **8**
Entiteterne og indbyrdes relationer 8
Normalisering **10**
1. Normalform 10
2. Normalform 10
3. Normalform 11
BCNF 12
SQL **13**
Oprettelse af tabeller 13
Forespørgsler 14
Udvidelser **16**
Selvvalgt sæde 16
Differentierede sædepriser 16
Distribution **16**
Datawarehouse **18**
Transaktions håndtering **18**
Webløsning **18**

Database Provider **19**
ODBC DB Provider 19
OLE DB Provider 19
SQL Server Provider 20
Applikations arkitektur **20**
Konklusion **21**
Litteratur liste **21**
Bilag **22**
Bilag 1 - ER-diagram 22
Bilag 2 - Afhængighedsdiagrammer 23
Bilag 3 - ER-diagram, normaliseret 24
Bilag 4 - SQL, oprettelse af tabeller 25
Bilag 5 - Skærmbillede til valg af sæde 29
Bilag 6 - Applikations Arkitektur 30

Problemformulering

Baggrunds beskrivelse

Danair skal have udviklet en database løsning til bookning a flysæder på deres afgange mellem København og destinationer i USA. Administrationen vil gerne kunne følge med i den samlede salgssum pr. Afgang, samt de totale gebyr omkostninger pr. Afgang. Derudover er det vigtigt at kunden får klar besked om afgang og siddeplads ved en booking.

Problemformulering

Danair er et flyselskab som udbyder pladser til deres flyruter. I flyselskabet kalder man internt hver solgt plads på en flyafgang for et solgt "flysæde". Danair flyver mellem København og destinationer i USA, primært New York.

Du skal ved booking sørge for at den enkelte kunde får oplyst, hvilken rute han/hun rejser (dvs dato, tid, navn på fly-afgang) og (og evt. hvilket sæde han/hun skal sidde på). Systemet kender blandt andet:

Flyafgange:	Dato, tid og navn på flyafgang. Pris pr sæde.
Fly-type:	Dvs Flytype med oplysninger om antal sæder i flyet.
Flysæder pr afgang:	Der findes oplysninger om flysæder pr flytype. Når fly-selskabet udbyder en konkret afgang (dato, tid og navn på flyafgang), anvendes disse oplysninger om antal sæder i flyet til at udbyde sæder i flyet. Når alle sæder er booked må der ikke sælges flere sæder (altså ingen over-booking).
Flybooking:	Navn og adresse på passageren. Om pladsen er bestilt eller betalt. Passagerer tlf nummer, pas.nr.

Diverse krav:
Det skal være muligt for administrationen i flyselskabet hele tiden at følge med i omsætningen på en konkret flyafgang. Her tænkes på den samlede salgssum for alle sæder.

Nogle lufthavne kræver "passagerergebyr"(lufthavnsafgift), hvor der skal betales ekstra gebyr pr passagerer der afrejser fra lufthavnen. Det skal være muligt for administrationen, at følge med i denne omkostning på en konkret flyafgang.

Opgaver:
Der skal udarbejdes en ER-model for en database til håndtering af ovenstående. Der bliver sikkert brug for at gøre nogle yderligere antagelser. Disse skal fremgå klart.

ER-modellen skal omformes til en relationel database.

Lav tabeldesign i henhold til designregler og BCNF.

Følgende skal implementeres i SQL:

1. CREATE TABLE-sætninger i SQL. Husk integritet..
2. Udarbejd SQL-sætninger til følgende forespørgsler:
3. Find alle fly-afgange på en given dato.
4. Vis alle afgange, som er fløjet mellem København og New York.
5. Find alle bookinger, som er bestilt/betalt af en bestemt passagerer.
6. Find alle bookinger pr postnummer.

Ovenstående skal laves.

Det følgende er forslag til udvidelser, som du kan medtage i din rapport. Medtag så mange som muligt og mindst to:

Udvidelser:

- At kunden får adgang til at aflyse/afbestille bestilt flysæde. Dette ved at der i forbindelse med bestillingen oplyses en bestillingskode/annulleringskode, som så senere kan anvendes ved annullering.
- Skitser løsning der giver kunden mulighed for selv at vælge hvilket konkret sæde han ønsker på en flyafgang.
- Pris pr sæde på en flyafgang varierer. Dvs. jo tættere man kommer på afgangsdatoen, jo dyrere bliver sædet. Hvordan håndteres dette?
- Skitser løsning til differentierede priser på flysæderne. Vinduespladser altid er eksempelvis 100 kr dyrere end øvrige pladser (eller businessclass).

Endvidere skal alle følgende 6 emner besvares:

Distribution
Danair overvejer at distribuere sin database geografisk, så der placeres en server i Danmark og en i New York. Danairs data tænkes placeret tættest på den server, som er fysisk tættest på en flyafgang.
Giv forslag til hvorledes denne opdeling af data skal fungere.
Synes du det er en god ide at opdele afgang-data på denne måde ?
Begrund dine valg og diskuter fordele og ulemper.

Data warehouse
Overvej hvorledes Danair kan stille ledelsesinformation til rådighed for den økonomiske ledelse. Hvilke dimensioner vil være relevante.

Transaktionshåndtering
Der kan være behov for særlige transaktioner (evt. distribuerede), som fx at flytte en booking reservationer fra én afgang til en anden. Diskuter hvilke problemer (og mulige løsninger) en sådan transaktion vil give anledning til.

Web-løsning
Det overvejes at tilbyde kunderne selvbetjening via en web-adgang til databasen. Beskriv kort hvilke krav dette vil stille til databaseadgangen, herunder transaktionshåndtering.

Database provider
Beskriv kort, hvilken forskel der er på Data providers: OLE DB provider, SQL Server provider og ODBC Data Provider. Diskutér fordele og ulemper ved hvert valg.

Applikationsprogrammering
Diskuter arkitektur for applikationer, som tilgår databasen. Belys med eksempler implementeret i C# og ADO.NET..)

Afgrænsninger

fly
Da jeg ikke har informationer om flylayout (sektioner, manglende sæder Pga. toilet osv) har jeg valgt at simplificere dette som et x,y skema hvor x er rækker og y er sæder. Desuden har jeg afgrænset flylayoutet til kun at have 1 etage.

Sædepladser
Jeg har vedtaget en navnestandard jævnfør ovenstående layout regel. Sæder navngives rækkepladsnummer. F.eks. 3-4 vil være række 3 fra cockpittet, plads Nr 4 fra venstre side af flyet.

Kunder
Da der i opgaveformuleringen ikke er beskrevet noget om specielle kundetyper, F.eks. Erhvervskunder med konto, er dette ikke medtaget. Dette kan let implementeres på et senere tidspunkt.

Udvidelser
Jeg har valgt at medtage følgende udvidelser fra start: Selvvalgt sæde og differentierede sædepriser.

Postkoder (ZIP) i USA
Jeg har valgt at antage, at amerikanske postkoder fungerer på samme måde som danske postnumre, dog med 5 cifre. Da jeg kun har lavet lettere research på emnet kan det være behæftet med fejl.

Valuta
Systemets arbejdsvaluta er danske kroner

Metode

Arbejdet med udvikling af databasen vil blive udført efter metoderne beskrevet i Database Håndbogen, af Joakim Dalby. Herunder anvendelse af ER-diagram og normalisering til BCNF.

Disposition

Foranalyse
Udvælgelse af entiteter

ER-Diagrammet
Entiteter og indbyrdes relationer

Normalisering
1-3NF
BCNF

SQL
Create Table
Forespørgsler

Udvidelser

Distribution
Opdeling af data.

Data warehouse
Ledelses information

Transaktionshåndtering
Særlige transaktioner

Webløsning
Sikkerhed

Database provider
Forskelle, fordele og ulemper

Applikationsprogrammering
Arkitektur.

Konklusion

Litteraturliste

Bilag

Foranalyse

Udvælgelse af entiteter

Under gennemlæsningen af opgaveformuleringen blev følgende entiteter fundet og valgt som relevante for opgavens løsning:

Flytype
Flytypen indeholder et typenavn og information om sædepladser.

Flyafgang
Flyafgang indeholder et afgangsnavn, samt informationer om dato, tid og destination.

Destination
Destination indeholder informationer om hvorfra og hvor til der rejses. Derudover er der informationer om pris pr. Sæde og eventuel afgift i afrejse lufthavnen.

Passager
Passager indeholder navn, bopæl, telefonnummer, pasnummer.

Kunde
Indeholder kontakt informationer for kunderne

PostBy
Indeholder postnumre og bynavne.

Booking
Booking indeholder et bookingnummer, information om hvilken afgang, samt hvilken kunde bookingen tilhører.

ER-Diagrammet

Udarbejdelsen af ER-diagrammet (Bilag 1) afstedkommer en del ændringer i de udvalgte entiteter. Nogle forsvinder, andre bliver ændret og nogle nye opstår, enten som følge af relationer, eller for at tilfredsstille krav fra problemformuleringen.

Entiteterne og indbyrdes relationer

Kunde
Entiteten Kunde indeholder kundens/passagerens kontaktoplysninger. De oprindelige entiteter, kunde og passager er slået sammen for at undgå redundans. Entitetsnavnet er valgt fordi en kunde ikke behøver at være en passager. Det kunne F.eks være en receptionist som booker billet for en medarbejder.

Kunde entiteten er opsplittet i sub tabeller over feltet Land for at tilgodese adresse og telefonnummer forskelligheder.

Entiteten er en svag entitet, da en kunde ikke kan eksistere uden enten, at have placeret en ordre eller være registreret på en booking.

Sub-entiteterne er ligeledes svage da de er afhængige af et postnummer og for USA's vedkommende også en stat.

Ordre
Entiteten holder oplysninger om ordredato, ordrenummer og om ordren er betalt, samt hvilken kunde, som har bestilt.

Ordre entiteten er svag da det ikke giver mening at have en ordre uden kunde, eller uden ordrelinjer (bookinger).

Booking
Entiteten booking er den reelle flysæde booking. Entiteten indeholder oplysninger om fly afgang, sædenummer, Kundenummer (Passager), samt pris ved bestillingstidspunktet.

Entiteten booking er svag, da man ikke kan lave en booking uden en flyafgang. Desuden er en ordre også påkrævet for at oprette en booking.

Flyafgang
Entiteten flyafgang samler flytype og destination med dato og tid. Entiteten er svag da en flyafgang ikke kan eksistere uden både fly og destination.

Destinationer
Destinationer indeholder afrejseby og destination, samt information om sædepris og eventuel lufthavns afgift.

En destination kan godt være oprettet uden at der er flyafgange til destinationen.

Flytype
Entiteten flytype indeholder en reference til flylayout. Da der i opgavebeskrivelsen ikke er beskrevet krav om at registrere andre data for et fly, er det den eneste attribut i tabellen.

Flytype entiteten er svag, da den ikke kan eksistere uden et flylayout.

Flylayout
Entiteten indeholder data til automatisk generering af et flylayout. Da der vil være rigtig mange opslag for at finde antal sæder, blandt andet til kontrol af overbooking er der et beregnet felt til antal sæder. Dette giver god mening da det må antages at antal sæder kun ændres i yderst sjældne tilfælde.

Et flylayout kan godt eksistere uden, at der er en flytype som benytter layoutet.

SpecialPriser
Entiteten indeholder navn og pris (tillæg) for specielle siddepladser, F.eks ekstra benlængde, vinduesplads, eller Business Class.

Der kan godt oprettes Specialpriser uden de er benyttet.

SpecialPrisPladser
Entiteten SpecialPrisPladser er resultatet af en mange-til-mange relation over flytype og specialpriser og er derfor automatisk en svag entitet, da der kræves reference til begge entiteter. Den indeholder information om sædeplads.

Valutakurs
Entiteten Valutakurs indeholder dagskurs. Der er en indirekte relation fra Ordre og Kunde til Valutakurs over fælles felterne Land ,OrdreDato og Dato.

Normalisering

1. Normalform

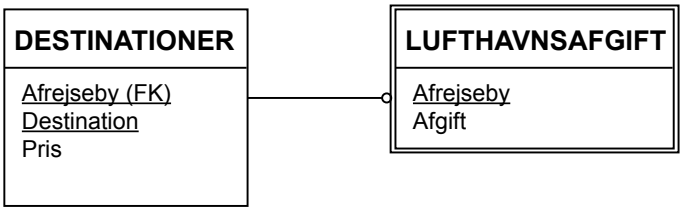
Da alle tabellerne har en primærnøgle og der ikke er gentagne grupper af felter i tabellerne, er de alle på 1. Normalform.

2. Normalform

Der er kun 2 tabeller med sammensat primærnøgle. De skal kontrolleres for 2. Normalform.

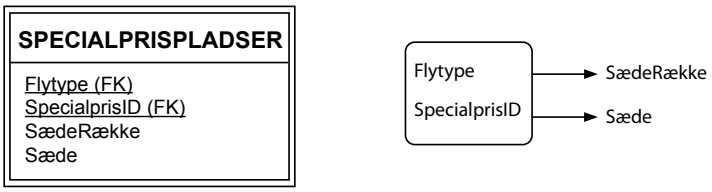
Destinationer

Da feltet afgift kun er afhængig af Afrejseby, som vist i afhængighedsdiagrammet, er tabellen ikke på 2. Normalform. Tabellen opsplittes derfor og determinanten Afrejseby bliver primærnøgle i den nye tabel. Entiteten Lufthavnsafgift er en svag entitet, da den er afhængig af Afrejsebyen.



Da begge tabeller kun indeholder 1 felt udenfor primærnøglen er de nu begge på 3. Normalform.

Specialprispladser

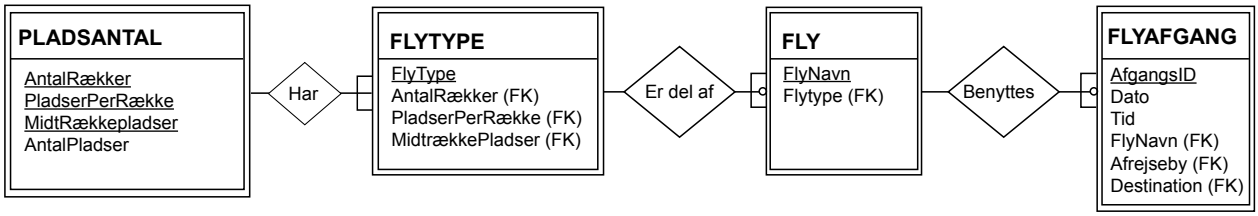


Da begge felter udenfor primærnøglen er afhængige af den fulde nøgle, er tabellen på 2. Normalform. Tabellen opfylder envidere kravene til 3. Normalform, da ingen af felterne udenfor primærnøglen er afhængige af hinanden. Derudover kan felterne SædeRække og Sæde betragtes som en sammensat værdi, en sædeplads og opfylder dermed definitionen om kun 1 felt som ikke indgår i primærnøglen.

3. Normalform

Der er 2 tabeller som ikke er på 3. Normalform. Denne normalform tager sig af transitive afhængigheder, og dem er der 2 af. Jeg starter med tabellen Flylayout.

Der eksisterer en transitiv afhængighed som vist i ovenstående afhængigheds diagram. Ingen af felterne er en del af primærnøglen og derfor skal der normaliseres. Antalpladser feltet er et beregnet felt og eksisterer kun for, at undgå en beregning af antal pladser, hver gang der skal tjekkes for overbooking. Primærnøglen LayoutID er en ”dummy” nøgle som ikke bringer nogen værdi til databasen udover at gøre det muligt at have flere layouts til en flytype. Jeg vælger derfor at lave en design ændring, som åbner op for nogle nye muligheder for senere udvidelser.

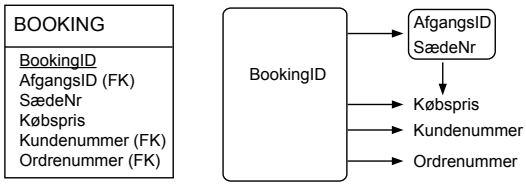


I tabellen Flyafgang er der kun ændret i en fremmednøgle, så den behøver ikke yderligere arbejde.

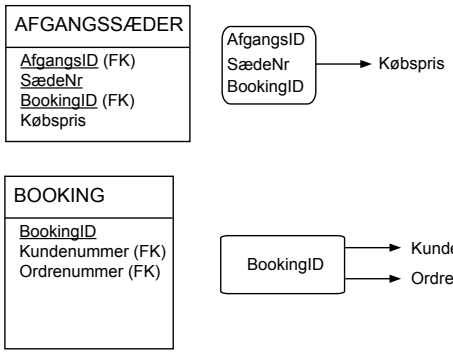
Tabellen Fly er ny. Tabellen indeholder specifikke fly og kan senere udvides med data om flyet, serviceplaner Osv. Tabellen er på 3. Normalform. Felterne fra den oprindelige tabel, Flylayout er flyttet op til tabellen Flytype. Feltet flylayout er elimineret og derved er den transitive afhængighed væk. Tabellen Flytype er på 3. Normalform, da der ikke er afhængighed mellem ikke nøgle felter.

Tabellen Pladsantal overholder 2. Normalform da AntalPladser er afhængig af den fulde nøgle. Da der kun er et felt udenfor primær nøglen er tabellen også på 3. Normalform.

Den næste tabel er Booking.



Den transitive afhængighed som vist i ovenstående diagram skal elimineres. Det gøres ved at opsplitte tabellen i 2 som vist herunder. Da et sæde i AfgangsSæder ikke kan eksistere uden at tilhøre en booking, kopieres feltet BookingID med over og bliver en del af nøglen i den nye tabel.



Alle tabeller opfylder nu 3. Normalform.

BCNF

BCNF er stærkere end 2. Og 3. Normalform og jeg kunne derfor have sprunget dem over. Dog har normaliseringsarbejdet givet anledning til yderligere overvejelser i forhold til designet, så derfor mener jeg ikke det har været helt spildt.

Som det kan ses i bilag 2 er der nu ingen determinanter som ikke er en kandidatnøgle. Derfor overholder tabellerne nu BCNF.

Det færdig normaliserede ER-Diagram kan ses i bilag 3.

SQL

Oprettelse af tabeller

Det fulde SQL script til oprettelse af tabellerne og deres relationer kan ses i bilag 4.

SQL server understøtter reelt ikke nedarvning. Det kan dog implementeres ved hjælp af nøgler og indekser. Feltet Land i tabellen Kunde er ikke en del af primærnøglen. Derfor er der oprettet et unikt indeks på felterne KundeID og Land. Derved kan felterne benyttes som fremmednøgler i de nedarvede tabeller.

Jeg har desuden benyttet et unikt indeks på tabellen Booking, for at undgå, at der kan bookes flere pladser på samme afgang, med samme passager.

Da der ikke må overbookes på en afgang har jeg oprettet en funktion som kontrollerer at der stadig er flere ledige pladser på en given afgang. Denne funktion anvendes i en CHECK constraint på tabellen AfgangsSæder. Funktionen ser således ud:

```
CREATE FUNCTION Danair.udf_SeatsAreAvailable(@AfgangsID NVARCHAR(7))
RETURNS BIT
AS
BEGIN
  DECLARE @Result BIT
  DECLARE @BookedSeats INT
  DECLARE @TotalNumberOfSeats INT
  SET @Result = 0

  SELECT @BookedSeats = COUNT(a.AfgangsID)
  FROM Danair.AfgangsSæder a
  WHERE a.AfgangsID = @AfgangsID

  SELECT @TotalNumberOfSeats = b.AntalPladser
  FROM Danair.PladsAntal b INNER JOIN Danair.Flytype c
    ON (b.AntalRækker = c.AntalRækker AND
      b.PladserPerRække = c.PladserPerRække AND
      b.MidtRækkePladser = c.MidtRækkePladser)
    INNER JOIN Danair.Fly d
      ON c.FlyType = d.FlyType
    INNER JOIN Danair.FlyAfgang e
      ON d.FlyNavn = e.FlyNavn
  WHERE AfgangsID = @AfgangsID
  IF (@BookedSeats <= @TotalNumberOfSeats)
  BEGIN
    SET @Result = 1
  END
  RETURN @Result
END
GO
```

Funktionen består af 2 select statements som tæller antallet af bookede sæder på en given afgang, samt henter værdien for det samlede antal pladser i det benyttede fly. Disse værdier sammenlignes og der returneres henholdsvis sandt hvis der er flere ledige pladser, eller falsk hvis der er fuldt booket.

Forespørgsler

Find alle flyafgange på en given dato:

```
SELECT *
  FROM Danair.FlyAfgang
  WHERE Dato = '20110220'
  ORDER BY 'Tid' ASC
```

Vis alle afgange, som er fløjet mellem København og New York:

```
SELECT * FROM danair.flyafgang
WHERE AfrejseBy = 'København'
  AND Destination = 'New York'
order by 'Dato' ASC, 'Tid' ASC
```

Find alle bookinger, som er bestilt/betalt af en bestemt passagerer:

Da jeg har åbnet op for muligheden for at en kunde kan bestille pladser for andre end sig selv, F.eks receptionisten i et firma som bestiller for medarbejdere, er jeg nødt til at vise forespørgslen fra både kundevinklen og fra passagervinklen. Forespørgslerne er stort set identiske. Den store forskel ligger i om der spørges på ordre niveau eller på booking niveau.

Kunden

```
SELECT Ordre.OrdreNummer, Booking.BookingID, AfgangsID, SædeNummer
FROM Danair.Ordre
  INNER JOIN Danair.Booking
    ON Danair.Ordre.OrdreNummer = Danair.Booking.OrdreNummer
  INNER JOIN Danair.AfgangsSæder
    ON Danair.Booking.BookingID = Danair.AfgangsSæder.BookingID
WHERE Danair.Ordre.KundeNummer = 1000 AND Betalt = 'false'
```

Passageren

```
SELECT Ordre.OrdreNummer, Booking.BookingID, AfgangsID, SædeNummer
FROM Danair.Ordre
  INNER JOIN Danair.Booking
    ON Danair.Ordre.OrdreNummer = Danair.Booking.OrdreNummer
  INNER JOIN Danair.AfgangsSæder
    ON Danair.Booking.BookingID = Danair.AfgangsSæder.BookingID
WHERE Danair.Booking.KundeNummer = 1000 AND Betalt = 'False'
```

Forskellen på de to forespørgsler ligger i WHERE sætningen. Den første fortæller hvilke ordrer fra en kunde, med hvilke bookinger, som er betalt, eller ikke betalt (true/false). Den anden forespørg-

sel viser hvilke bookinger en passager er registreret på, hvor ordren er betalt, eller ikke er betalt.

Som systemet er bygget op, er den første forespørgsel nok den mest sigende, da betaling sker på ordre niveau.

Find alle bookinger pr postnummer:

```
SELECT Danmark.PostNummer, AfgangsID, Booking.BookingID
FROM Danair.AfgangsSæder
  INNER JOIN Danair.booking
  ON Danair.AfgangsSæder.bookingID = Danair.Booking.BookingID
  INNER JOIN Danair.Kunde
  on Danair.Booking.KundeNummer = Danair.Kunde.KundeNummer
  INNER JOIN Danair.Danmark
  ON Danair.Kunde.KundeNummer = Danair.Danmark.KundeNummer
  INNER JOIN Danair.PostBy
  ON Danair.Danmark.PostNummer = Danair.PostBy.PostNummer
UNION
SELECT Usa.PostNummer, AfgangsID, Booking.BookingID
FROM Danair.AfgangsSæder
  INNER JOIN Danair.Booking
  ON Danair.AfgangsSæder.BookingID = Danair.Booking.BookingID
  INNER JOIN Danair.Kunde
  ON Danair.Booking.KundeNummer = Danair.Kunde.KundeNummer
  INNER JOIN Danair.Usa
  ON Danair.Kunde.KundeNummer = Danair.Usa.KundeNummer
  INNER JOIN Danair.PostBy
  ON Danair.Usa.PostNummer = Danair.PostBy.PostNummer
ORDER BY
  PostNummer
  ,AfgangsID ASC
  ,BookingID ASC
```

Omsætning på konkret flyafgang
For at administrationen kan følge omsætningen på en konkret flyafgang kan følgende forespørgsel benyttes:

```
SELECT AfgangsID, SUM(KøbsPris) AS 'Total Omsætning'
FROM Danair.AfgangsSæder
WHERE AfgangsID = 'cph0001'
GROUP BY AfgangsID
```

Hvis administrationen ønsker en liste med total omsætning pr flyafgang skal man blot udelade WHERE sætningen.

Omkostning på konkret flyafgang
For at følge de totale omkostninger (lufthavns afgif) på en konkret flyafgang benyttes følgende forespørgsel. Igen kan WHERE sætningen udelades for at få en liste.

```
SELECT b.AfgangsID, SUM(Afgift*Antal) as Omkostninger
FROM (SELECT AfgangsID, COUNT(AfgangsID) Antal
    from Danair.AfgangsSæder
    GROUP BY AfgangsID) a
  INNER JOIN Danair.FlyAfgang b
    ON a.AfgangsID = b.AfgangsID
  INNER JOIN Danair.LufthavnsAfgift c
    ON b.AfrejseBy = c.AfrejseBy
WHERE b.AfgangsID = 'ny0003'
GROUP BY b.AfgangsID
```

Udvidelser

Selvvalgt sæde

Jeg har valgt at definere et flylayout ved et antal rækker og kolonner. Derved kan jeg programatisk generere et skema med flysæder i form af knapper. Disse flysæder kan så farvekodes efter specialpriser Osv. Brugeren, eller kunden kan så selv klikke på et sæde for at bestille en plads. Der skal selfølgelig også være mulighed for at bestille et tilfældigt sæde udfra nogle parametre, F.eks om det skal være business class eller andre specielle forhold, som er opgivet i tabellen specialpriser.

Bilag 5 viser et eksempel på hvordan sådan et skærmbillede kunne se ud. Når et flysæde allerede er bestilt markeres det med rød og knappen bliver inaktiv. Derved er det kun muligt at vælge ledige sæder.

Differentierede sædepriser

For at gøre det muligt at lave differentierede sædepriser, har jeg oprettet tabellen SpecialPriser. Denne tabel indeholder navn på specialprisen F.eks Business Class, eller vinduesplads, samt merprisen. Tabellen SpecialPrisPladser samler specialpriser med et sæde (sæderække, sæde) i et givent fly. Jeg har her antaget at inddelingen af specialprispladser ikke ændres fra afgang til afgang, men følger flyet.

Sammen med løsningen for Selvvalgt sæde er det muligt dels at visualisere ved hjælp af farvekoder eller lignende, samt at lave opslag for at finde merprisen for et givent sæde.

Distribution

At distribuere sin datase over flere lokationer er en god ide. Det beskytter blandt andet data i tilfælde af nedbrud, da data vil være fuldt repræsenteret på de øvrige lokationer. I en distribueret database fungerer distribueringen transparent.

For at distribuere Danairs data således at data er placeret tættest på en flyafgang er vi nødt til at kigge på hvor de forskellige tabeller hører til. Jeg har valgt at opdele ud fra 3 lokationer, hvor Hovedkontor og København er samme fysiske server, men set fra 2 forskellige vinkler.

	Hovedkontor	København	New York
Fly	X		
FlyType	X		
PladsAntal	X		
SpecialPrisPladser	X		
SpecialPriser	X		
FlyAfgang		X	X
Destinationer		X	X
LufthavnsAfgift		X	X
Afgangssæder		X	X
Booking		X	X
Ordre		X	X
Kunde		X	X
Danmark		X	
Usa			X
PostBy		X	X
Stat			X
Valutakurs	X		

Som det kan ses af ovenstående tabel giver det en horisontal opsplitning af data på de fleste tabeller. Kun tabellerne Danmark, Usa og Stat, hører entydigt til på en bestemt lokation. Jeg har antaget at selskabet arbejder udfra en samlet flypark og derfor hører de tabeller som omhandler fly, til hovedkontoret. Der sker formodentlig ikke så mange ændringer på disse tabeller. Ligeledes antager jeg at hovedkontoret tager sig af Valutakurser.

Fordele
Fordelen ved at distribuere Danairs data som vist, er at begge lokationer kan fungere uafhængigt af hinanden. Hvis den ene lokations database server pludselig fejler, kan der stadig oprettes kunder og ordre Osv hos den anden lokation.

Ulemper
Af ulemper kan nævnes ekstra omkostninger til hardware, samt ekstra administrations omkostninger. Endnu en ulempe ved sådan en løsning er problemer med primær nøgler, specielt tilfælde hvor IDENTITY felter indgår i primærnøglen. Dette kan dog løses ved at tildele hver lokation et interval af nøgler. F.eks København 1-100000 - New York 100001-200000.

Datawarehouse

Information er vigtig. Især når det gælder økonomi. For Danairs økonomiske ledelse er der mange spørgsmål som kunne være interessante at have svar på. De fleste spørgsmål involverer historisk data. Dette kan et DataWarehouse løse. Jeg vil her kort præsentere et par af dem:

- Hvor fyldte er vores afgange?
- Vedligeholdelses statistik?

Historiske data er vigtige blandt andet fordi de viser tendenser. Hvis man F.eks ser på antallet af passagerer per afgang over tid, vil man kunne aflæse om man skal ud og investere i større fly, eller oprette flere afgange.

Ved at registrere økonomiske data om vedligeholdelse af flyparken vil man være bedre i stand til at vurdere hvornår et fly ikke længere er rentabelt at vedligeholde og dermed have et grundlag for en investeringsplan i forhold til flyparken.

Transaktions håndtering

Der er meget som kan gå galt når man opdaterer eller indsætter data i en database. Derfor kan man med fordel benytte transaktioner, da en transaktion som fejler under udførelsen vil spole alle handlingerne tilbage ved en fejl. Med udgangspunkt i situationen hvor en booking skal flyttes fra en afgang til en anden vil jeg her redegøre for en alvorlig fejlkilde, samt hvordan man løser dette og andre problemer med transaktioner, herunder distribuerede transaktioner.

Den største fejlkilde i denne transaktion er antallet af ledige pladser på den afgang man flytter til, da der ikke må overbookes. En Check Constraint kontrollerer om der er flere ledige pladser.

En distribueret transaktion er på mange måder lig en almindelig transaktion på en enkelt server. Forskellen ligger i hvordan SQL Server håndterer transaktionen. En Distribueret transaktion bliver afviklet i 2 faser, Prepare fasen og Commit fasen. Når transaktions manageren får en commit request, sender den en prepare kommando ud til alle servere som er involveret i transaktionen. Serverne gør sig klar og melder tilbage til transaktions manageren at de er klar. Herefter sendes en Commit kommando til alle serverne. Først når transaktionsmanageren har modtaget besked om succes fra alle serverne, gives klienten besked om en vellykket transaktion. Såfremt en af serverne rapporterer om fejl under behandlingen af transaktionen, sender transaktionsmanageren en Rollback kommando til serverne og fejl rapporteres til klienten.

Webløsning

Når man vælger at tilbyde sine kunder selvbetjening via internettet stiller det store krav til data sikkerheden. Det er jo ikke alene lovlydige kunder som får adgang, men reelt alle med en internet forbindelse. For at tilfredsstille disse krav til datasikkerhed er det vigtigt at Web applikationen benytter en database bruger som har begrænset adgang til databasen. Derudover bør Web.config, hvor connection strengen ofte er gemt, være krypteret.

Der er potentielt mange samtidige brugere i en Web applikation. Derfor bør transaktionerne være så små som muligt, da transaktioner låser data mens de kører. Input validering er også vigtig. Alle data bør være indsamlet og valideret inden en transaktion åbnes og data sendes til sql serveren, da en transaktion ellers vil fejle. ASP.NET understøtter både serverside validering og med hjælp fra javascript også clientside validering.

Database Provider

ODBC DB Provider

ODBC er en gammel teknologi, som bruges til generel database adgang, hvor der ikke er mulighed for at bruge enten OLE DB eller specifikke providere. ODBC virker gennem en specifik driver til datakilden. Provideren er 3-tier i arkitektur.

Fordele
ODBC kan benyttes til ældre data kilder som ikke understøttes af OLE DB, eller de mere specifikke ADO.NET dataprovidere.

Ulemper
Der skal installeres ODBC driver på klienten.

ODBC er lagdelt og kan derfor give performance problemer.

OLE DB Provider

OLE DB provideren bruges til at forbinde til ældre versioner af SQL server, samt andre databaser som ikke har egen .net provider. OLE DB data provideren arbejder gennem 2 ekstra lag for at nå databasen. Dette bevirker at den ikke er nær så hurtig som F.eks sql server.net provideren som snakker direkte med sql serveren.

Fordele
Dataprovideren kan ”snakke” med datakilder som ikke har en specifik .net provider.

Ulemper
OLE DB er lagdelt af struktur. Dette bevirker en forringelse i performance.

SQL Server Provider

SQL Server .NET Data provideren er specifikt designet til at snakke med SQL Server. Da dataprovideren ikke er lagdelt, men snakker direkte med serveren, performer den generelt hurtigere end både ODBC og OLE DB.

Fordele
Dataprovideren snakker direkte med database serveren

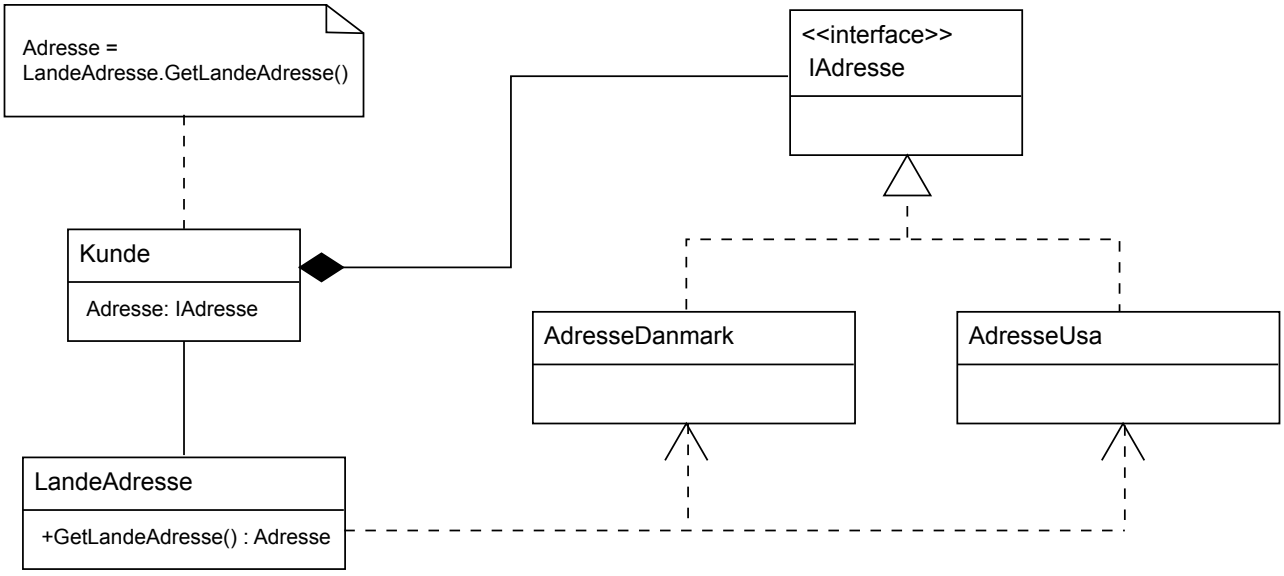
Performance er bedre da arkitekturen ikke er lagdelt

Ulemper
Dataprovideren kan ikke snakke med SQL Server 6.5 og ældre versioner.

Applikations arkitektur

Jeg har valgt at illustrere den del af applikations arkitekturen som omhandler kunder og adresser i en lettere simplificeret form.

Adresser har forskelligt format alt efter hvilket land adressen befinder sig i. Derfor har jeg valgt en løsning hvor der oprettes en tabel pr land til adressen. For at få det rigtige objekt og dermed den rigtige tabel har jeg anvendt design mønstret Factory Method som vist herunder.



Kunden bestemmer entydigt adressetypen via feltet Land. Derfor er det Kunden som beder LandeAdresse om en adresse.

Konklusion

Da jeg var færdig med projektet blev jeg opmærksom på en problemstilling, som jeg lige kort vil redegøre for. Når en kunde booker et flysæde får han en beregnet pris
(pris + specialpladspris). Problemet opstår når kunden vælger at betale senere end ordredatoen. Hvis en eller flere priser er ændret i mellemtiden, vil et prisopslag på betalingsdagen være ændret i forhold til bestillingsdagen. Problemet er løst intermistisk med feltet Købspris på entiteten Booking, men det bøjer normaliseringsreglerne da det nemt kunne slåes op. I en alternativ, og måske bedre, løsning kunne man oprette en tabel med dagspriser, så man kunne slå prisen op på en given dag.

Litteratur liste

Kilde Nr. 1
Dalby, Joakim (1994). ”Database Håndbogen”, Samfundslitteratur, 2. udgave 1994.

Kilde Nr. 2
Agerwal, Vidya Vrat, et al. (2008). ”Beginning C# 2008 Databases”, APress 2008.

Kilde Nr. 3
Thernstrom, Tobias, et al. (2008). ”MCTS Self-Paced Training Kit (Exam 70-433)”, MS Press 2008.

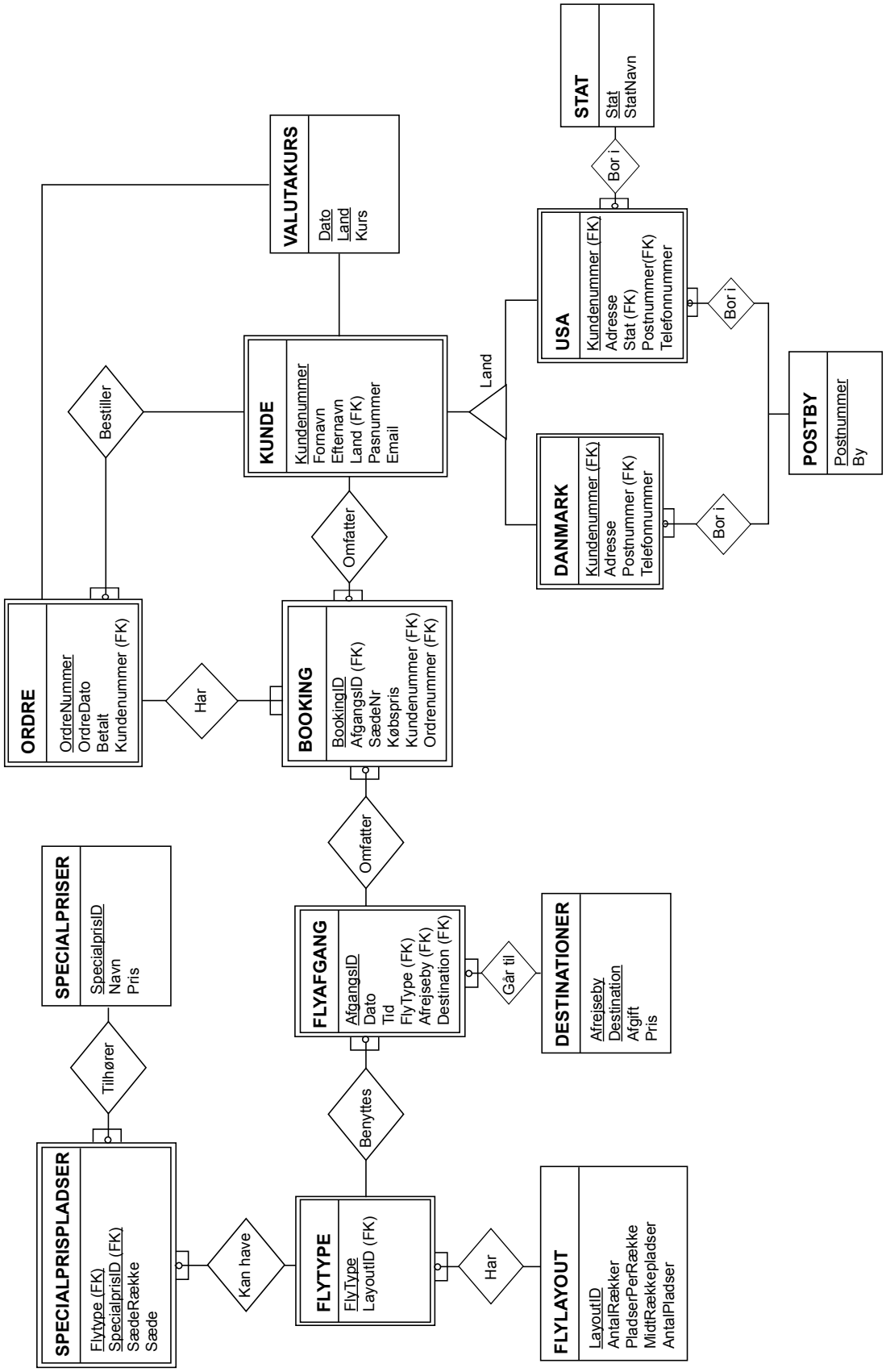
Kilde Nr 4
Whitehorn & Marklyn (2003). ”Relationelle Databaser”, Libris 2003.

Kilde Nr. 5
http://en.wikipedia.org/wiki/Address_(geography). ”Adresse format”, Wiki.

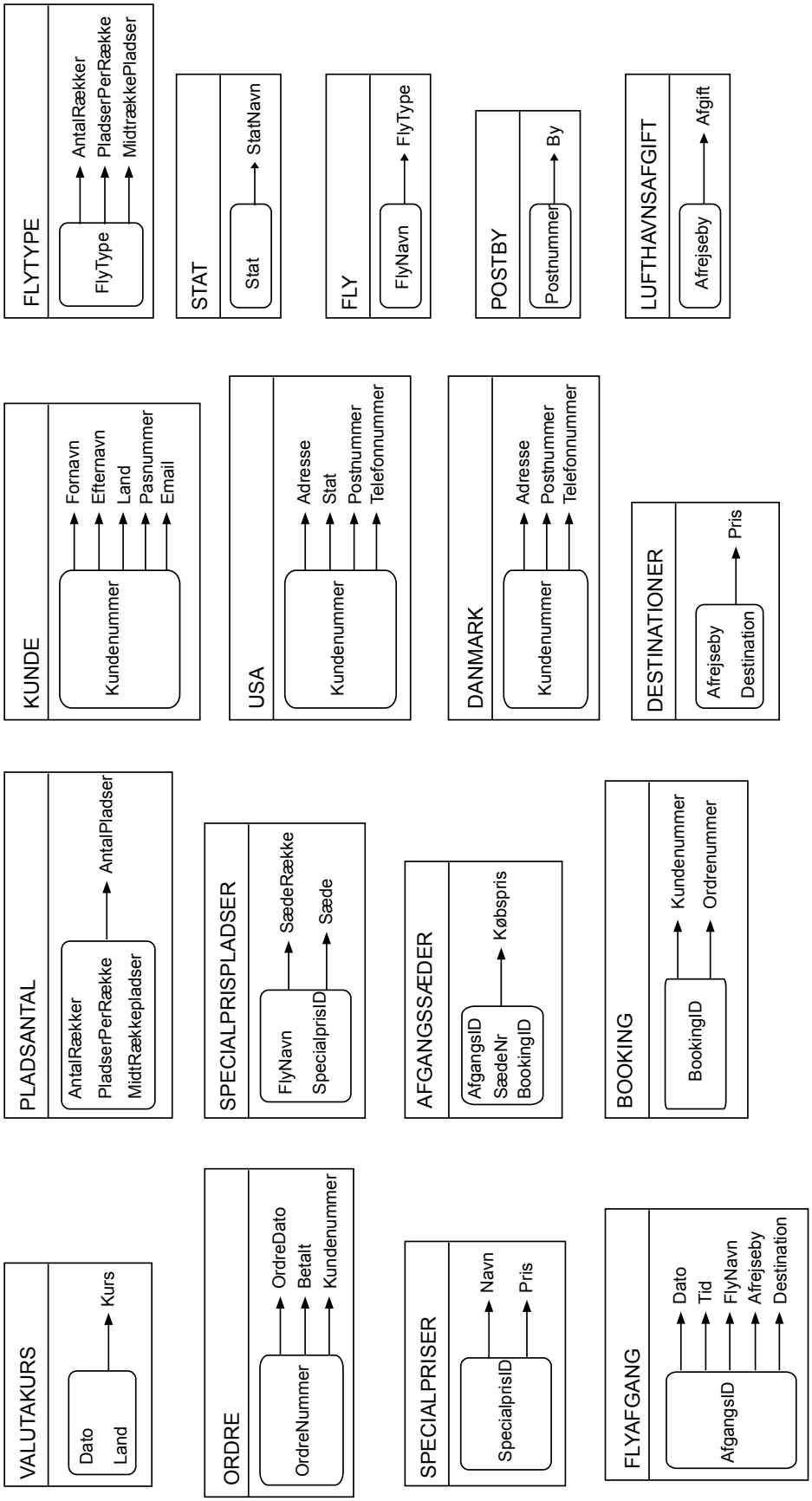
Kilde Nr. 6
http://msdn.microsoft.com

Bilag

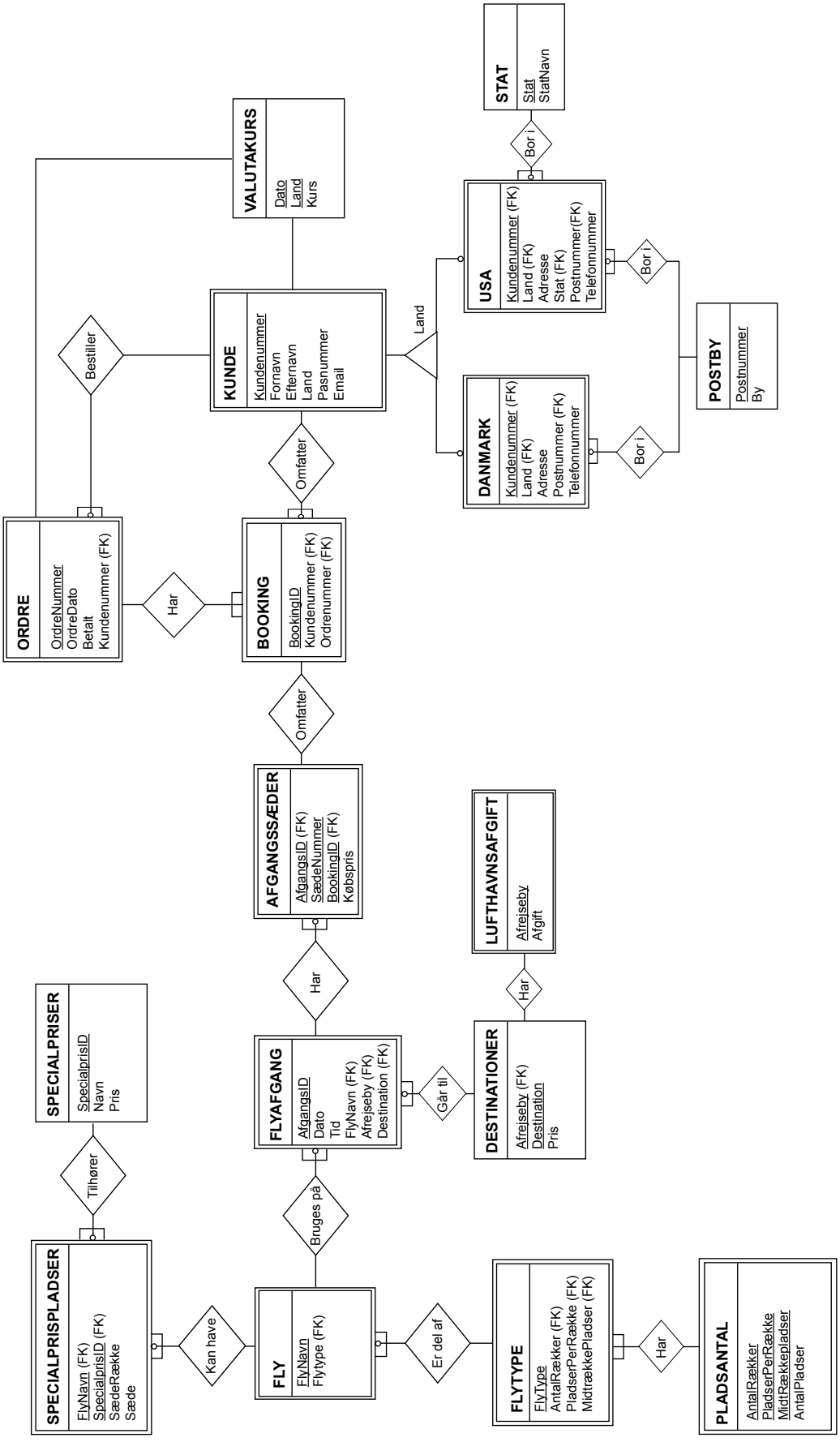
Bilag 1 - ER-diagram



Bilag 2 - Afhængighedsdiagrammer



Bilag 3 - ER-diagram, normaliseret



Bilag 4 - SQL, oprettelse af tabeller

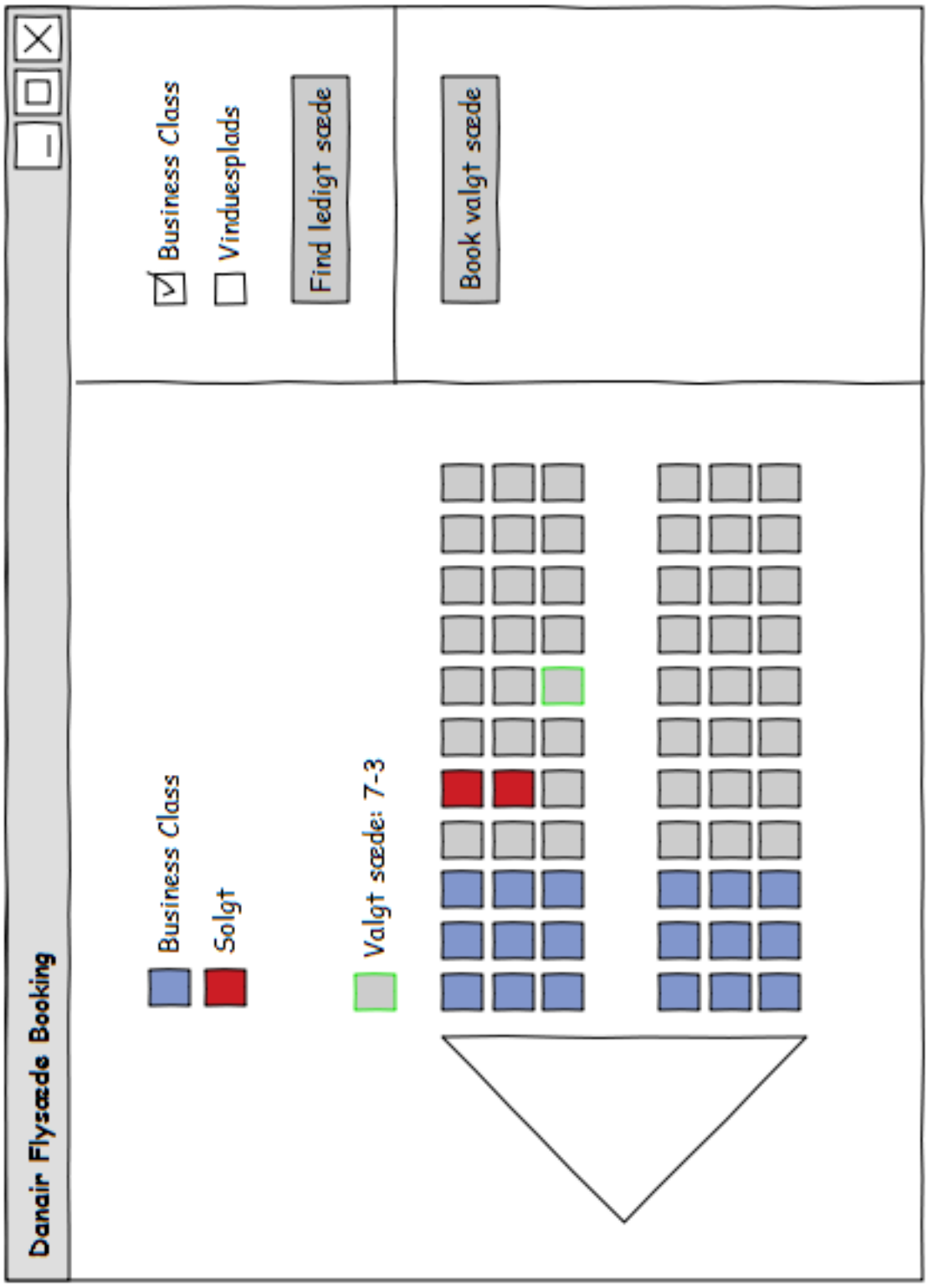
```
USE MASTER
GO
CREATE DATABASE Danair
GO

USE Danair
GO

CREATE SCHEMA Danair
GO

CREATE TABLE Danair.PostBy (
  PostNummer INT PRIMARY KEY
  ,ByNavn NVARCHAR(50) NOT NULL
);

CREATE TABLE Danair.Stat (
  Stat NCHAR(2) PRIMARY KEY
  ,StatNavn NVARCHAR(50) NOT NULL
);


CREATE TABLE Danair.ValutaKurs (
  Dato DATE NOT NULL
  ,Land NVARCHAR(20) NOT NULL
  ,Kurs DECIMAL(10,4) NOT NULL
  ,CONSTRAINT PK_Valutakurs PRIMARY KEY(Dato, Land)
);

CREATE TABLE Danair.Kunde (
  KundeNummer INT IDENTITY(1000,1) PRIMARY KEY
  ,ForNavn NVARCHAR(50) NOT NULL
  ,EfterNavn NVARCHAR(50) NOT NULL
  ,Land NVARCHAR(20) NOT NULL
  ,PasNummer INT
  ,Email NVARCHAR(50)
  ,CONSTRAINT UN_KundeLand UNIQUE(KundeNummer, Land)
);

CREATE TABLE Danair.Danmark (
  KundeNummer INT PRIMARY KEY
  ,Land NVARCHAR(20) NOT NULL
  ,Adresse NVARCHAR(50) NOT NULL
  ,PostNummer INT NOT NULL
  ,TelefonNummer NVARCHAR(8)
  ,CONSTRAINT FK_DKKunde FOREIGN KEY(KundeNummer, Land)
    REFERENCES Danair.Kunde(KundeNummer, Land)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  ,CONSTRAINT CK_DKLand CHECK(Land = 'Danmark')
  ,CONSTRAINT FK_PostNummer FOREIGN KEY(PostNummer)
    REFERENCES Danair.PostBy(PostNummer)
      ON UPDATE CASCADE
  ,CONSTRAINT CK_DKTelefon CHECK (TelefonNummer LIKE '[0-9][0-9][0-9][0-9][0-9]
[0-9][0-9][0-9]')
);
```

```
CREATE TABLE Danair.Usa (
  KundeNummer INT PRIMARY KEY
  ,Land NVARCHAR(20) NOT NULL
  ,Adresse NVARCHAR(50) NOT NULL
  ,Stat NCHAR(2) NOT NULL
  ,PostNummer INT NOT NULL
  ,TelefonNummer NVARCHAR(8)
  ,CONSTRAINT FK_USKunde FOREIGN KEY(KundeNummer, Land)
    REFERENCES Danair.Kunde(KundeNummer, Land)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  ,CONSTRAINT CK_USLand CHECK(Land = 'Usa')
  ,CONSTRAINT FK_USPostNummer FOREIGN KEY(PostNummer)
    REFERENCES Danair.PostBy(PostNummer)
      ON UPDATE CASCADE
  ,CONSTRAINT FK_Stat FOREIGN KEY(Stat)
    REFERENCES Danair.Stat(Stat)
      ON UPDATE CASCADE
  ,CONSTRAINT CK_UStelefon CHECK (TelefonNummer LIKE '[0-9][0-9][0-9]-[0-9][0-9]
[0-9][0-9]')
);

CREATE TABLE Danair.Ordre (
  OrdreNummer INT IDENTITY PRIMARY KEY
  ,OrdreDato DATE NOT NULL
  ,Betalt BIT DEFAULT '0' NOT NULL
  ,KundeNummer INT NOT NULL
    REFERENCES Danair.Kunde(KundeNummer)
      ON DELETE NO ACTION
      ON UPDATE CASCADE
);

CREATE TABLE Danair.LufthavnsAfgift (
  AfrejseBy NVARCHAR(50) PRIMARY KEY
  ,Afgift SMALLMONEY NOT NULL
);

CREATE TABLE Danair.Destinationer (
  AfrejseBy NVARCHAR(50)
  ,Destination NVARCHAR(50)
  ,Pris SMALLMONEY NOT NULL
  ,CONSTRAINT PK_Destinationer PRIMARY KEY(AfrejseBy, Destination)
  ,CONSTRAINT FK_Destinationer FOREIGN KEY(AfrejseBy)
    REFERENCES Danair.LufthavnsAfgift(AfrejseBy)
);

CREATE TABLE Danair.PladsAntal (
  AntalRækker TINYINT
  ,PladserPerRække TINYINT
  ,MidtRækkePladser TINYINT
  ,AntalPladser AS AntalRækker * (PladserPerRække + MidtRækkePladser) PERSISTED
NOT NULL
  ,CONSTRAINT PK_PladsAntal PRIMARY KEY(AntalRækker, PladserPerRække,
MidtRækkePladser)
);
```

```
CREATE TABLE Danair.FlyType (
  FlyType NVARCHAR(20) PRIMARY KEY
  ,AntalRækker TINYINT NOT NULL
  ,PladserPerRække TINYINT NOT NULL
  ,MidtRækkePladser TINYINT NOT NULL
  ,CONSTRAINT FK_Flytype FOREIGN KEY(AntalRækker, PladserPerRække,
MidtRækkePladser)
    REFERENCES Danair.PladsAntal(AntalRækker, PladserPerRække, MidtRækkePladser)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);

CREATE TABLE Danair.Fly (
  FlyNavn NVARCHAR(20)PRIMARY KEY
  ,FlyType NVARCHAR(20) NOT NULL
    REFERENCES Danair.Flytype(FlyType)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);

CREATE TABLE Danair.SpecialPriser (
  SpecialPrisID INT IDENTITY PRIMARY KEY
  ,Navn NVARCHAR(30) NOT NULL
  ,Pris SMALLMONEY NOT NULL
);

CREATE TABLE Danair.SpecialPrisPladser (
  FlyNavn NVARCHAR(20)
  ,SpecialPrisID INT
  ,SædeRække TINYINT NOT NULL
  ,Sæde TINYINT NOT NULL
  ,CONSTRAINT PK_SpecialPrisPladser PRIMARY KEY(FlyNavn, SpecialPrisID)
  ,CONSTRAINT FK_SpecialPris FOREIGN KEY(SpecialPrisID)
    REFERENCES Danair.SpecialPriser(SpecialPrisID)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  ,CONSTRAINT FK_Fly FOREIGN KEY(FlyNavn)
    REFERENCES Danair.Fly(FlyNavn)
      ON DELETE CASCADE
      ON UPDATE CASCADE
);

CREATE TABLE Danair.Flyafgang (
  AfgangsID NVARCHAR(7) PRIMARY KEY
  ,Dato DATE NOT NULL
  ,Tid TIME NOT NULL
  ,FlyNavn NVARCHAR(20) NOT NULL
  ,AfrejseBy NVARCHAR(50) NOT NULL
  ,Destination NVARCHAR(50) NOT NULL
  ,CONSTRAINT FK_BenyttetFly FOREIGN KEY(FlyNavn)
    REFERENCES Danair.Fly(FlyNavn)
      ON DELETE NO ACTION
      ON UPDATE CASCADE
  ,CONSTRAINT FK_Destination FOREIGN KEY(AfrejseBy, Destination)
    REFERENCES Danair.Destinationer(AfrejseBy, Destination)
      ON DELETE NO ACTION
      ON UPDATE NO ACTION
);
```

```
CREATE TABLE Danair.Booking (
  BookingID INT IDENTITY PRIMARY KEY
  ,KundeNummer INT NOT NULL
  ,OrdreNummer INT NOT NULL
  ,CONSTRAINT FK_Passager FOREIGN KEY(KundeNummer)
    REFERENCES Danair.Kunde(KundeNummer)
      ON UPDATE CASCADE
  ,CONSTRAINT FK_Ordre FOREIGN KEY(OrdreNummer)
    REFERENCES Danair.Ordre(OrdreNummer)
  ,CONSTRAINT UN_KundeBooking UNIQUE (KundeNummer, OrdreNummer)
);

CREATE TABLE Danair.AfgangsSæder (
  AfgangsID NVARCHAR(7)
  ,SædeNummer NVARCHAR(6)
  ,BookingID INT
  ,KøbsPris SMALLMONEY NOT NULL
  ,CONSTRAINT PK_AfgangsSæder PRIMARY KEY(AfgangsID, SædeNummer, BookingID)
  ,CONSTRAINT FK_Afgang FOREIGN KEY(AfgangsID)
    REFERENCES Danair.FlyAfgang(AfgangsID)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  ,CONSTRAINT FK_Booking FOREIGN KEY(BookingID)
    REFERENCES Danair.Booking(BookingID)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  ,CONSTRAINT CK_SeatsAvailable CHECK (Danair.udf_SeatsAreAvailable(AfgangsID) =
1)
);
```

Bilag 5 - Skærmbillede til valg af sæde



Bilag 6 - Applikations Arkitektur

```
using System;
using System.Data.SqlClient;

namespace Danair
{

  /*************************************************************************
   *              Selve Programmet                 *
   *************************************************************************
/


  class Program
  {
    static void Main(string[] args)
    {
      Kunde.ListKunder();

      Console.WriteLine("Indtast et kundenummer: ");
      int kundenummer = int.Parse(Console.ReadLine());

      Kunde Kunde1 = new Kunde(kundenummer);
      Kunde1.GetAdresse();
    }

  }

/*************************************************************************
 *              Interface IAdresse              *
 *************************************************************************/

  public interface IAdresse
  {
  }

/*************************************************************************
 *              Abstrakt Klasse Adresse           *
 *************************************************************************/

  public abstract class Adresse
  {
    public abstract override string ToString();
  }
```

```
/*************************************************************************
 *             Klassen AdresseDanmark            *
 *************************************************************************/

  public class AdresseDanmark : Adresse, IAdresse
  {
    private int KundeNummer;
    private string Land;
    public string Adresse { get; set; }
    public int PostNummer { get; set; }
    public string TelefonNummer { get; set; }

    public AdresseDanmark(int KundeNummer, string Land)
    {
      this.KundeNummer = KundeNummer;
      this.Land = Land;

      SqlConnection conn = new SqlConnection(@"
        server = .\sqlexpress; integrated security = true; database = Danair");

      string sqlstring = @"SELECT Adresse, PostNummer, TelefonNummer FROM
Danair." + Land + " WHERE KundeNummer = " + this.KundeNummer;

      SqlCommand cmd = new SqlCommand(sqlstring, conn);

      try
      {
        conn.Open();
        SqlDataReader rdr = cmd.ExecuteReader();

        while (rdr.Read())
        {
          Adresse = rdr.GetValue(0).ToString();
          PostNummer = int.Parse(rdr.GetValue(1).ToString());
          TelefonNummer = rdr.GetValue(2).ToString();

          Console.WriteLine("kunde nummer: {0} har følgende adresse\n{1}",
            KundeNummer, this.ToString());
        }
      }
      catch (SqlException ex)
      {
        Console.WriteLine("Error: " + ex.ToString());
      }
      finally
      {
        conn.Close();
      }
    }

    public override string ToString()
    {
      return Adresse + "\n" + PostNummer + "\n" + TelefonNummer;
    }
  }
```

```
/*************************************************************************
 *              Klassen AdresseUsa                *
 *************************************************************************/

public class AdresseUsa : Adresse, IAdresse
  {
    private int KundeNummer;
    private string Land;
    public string Adresse { get; set; }
    public string Stat { get; set; }
    public int PostNummer { get; set; }
    public string TelefonNummer { get; set; }

    public AdresseUsa(int KundeNummer, string Land)
    {
      this.KundeNummer = KundeNummer;
      this.Land = Land;

      SqlConnection conn = new SqlConnection(@"
        server = .\sqlexpress; integrated security = true; database = Danair");

      string sqlstring = @"SELECT Adresse, Stat, PostNummer, TelefonNummer FROM
        Danair." + Land + " WHERE KundeNummer = " + this.KundeNummer;

      SqlCommand cmd = new SqlCommand(sqlstring, conn);

      try
      {
        conn.Open();
        SqlDataReader rdr = cmd.ExecuteReader();

        while (rdr.Read())
        {
          Adresse = rdr.GetValue(0).ToString();
          Stat = rdr.GetValue(1).ToString();
          PostNummer = int.Parse(rdr.GetValue(2).ToString());
          TelefonNummer = rdr.GetValue(3).ToString();

          Console.WriteLine("kunde nummer: {0} har følgende adresse\n{1}",
            KundeNummer, this.ToString());
        }
      }
      catch (SqlException ex)
      {
        Console.WriteLine("Error: " + ex.ToString());
      }
      finally
      {
        conn.Close();
      }
    }

    public override string ToString()
    {
      return Adresse + "\n" + Stat + "\n" + PostNummer + "\n" + TelefonNummer;
    }
  }
```

```
/*************************************************************************
 *              Klassen Kunde                *
 *************************************************************************/

public class Kunde
  {
    public int KundeNummer { get; set; }
    public string ForNavn { get; set; }
    public string EfterNavn { get; set; }
    public string Land { get; set; }
    public int? PasNummer { get; set; }
    public string Email { get; set; }

    public Kunde(int KundeNummerI)
    {
      SqlConnection conn = new SqlConnection(@"
        server = .\sqlexpress; integrated security = true; database = Danair");

      string sqlstring = @"SELECT * FROM Danair.Kunde WHERE KundeNummer = " +
        KundeNummerI;

      SqlCommand cmd = new SqlCommand(sqlstring, conn);

      try
      {
        conn.Open();
        SqlDataReader rdr = cmd.ExecuteReader();

        while (rdr.Read())
        {
          KundeNummer = int.Parse(rdr.GetValue(0).ToString());
          ForNavn = rdr.GetValue(1).ToString();
          EfterNavn = rdr.GetValue(2).ToString();
          Land = rdr.GetValue(3).ToString();
          Email = rdr.GetValue(5).ToString();

          Console.WriteLine("kunde objekt oprettet med værdierne {0} {1} {2} {3}
            {4} {5}", rdr.GetValue(0), rdr.GetValue(1), rdr.GetValue(2),
            rdr.GetValue(3), rdr.GetValue(4), rdr.GetValue(5));
        }
      }
      catch (SqlException ex)
      {
        Console.WriteLine("Error: " + ex.ToString());
      }
      finally
      {
        conn.Close();
      }
    }

    public object GetAdresse()
    {
      LandeAdresse LandeAdresse = new LandeAdresse();
      IAdresse adresse;

      return adresse = LandeAdresse.GetLandeAdresse(Land, KundeNummer);
    }
```

```
    public static void ListKunder()
    {
      SqlConnection conn = new SqlConnection(@"
        server = .\sqlexpress; integrated security = true; database = Danair");

      string sqlstring = @"SELECT * FROM Danair.Kunde";

      SqlCommand cmd = new SqlCommand(sqlstring, conn);

      try
      {
        conn.Open();
        SqlDataReader rdr = cmd.ExecuteReader();

        while (rdr.Read())
        {
          Console.WriteLine("{0} {1} {2} {3}", rdr.GetValue(0), rdr.GetValue(1),
            rdr.GetValue(2), rdr.GetValue(3));
        }
      }
      catch (SqlException ex)
      {
        Console.WriteLine("Error: " + ex.ToString());
      }
      finally
      {
        conn.Close();
      }
    }
  }
}
```