2013

Kjørehjelperen

Produktdokumentasjon

Høgskolen i Oslo og Akershus

Henrik Hermansen og Lars Smeby
Gruppe 8
26.05.2013

Forord

Produktbeskrivelsen inneholder en beskrivelse av ”Kjørehjelperen”. De første kapitlene redegjør for ulike teknologier og rammeverk som er benyttet. Deretter gjennomgås programmet i detalj. Her går vi inn på hvordan programmet er bygget opp, hvordan kommunikasjonen foregår internt i programmet og hvilke datastrukturer som benyttes. Til slutt ser vi på det grafiske brukergrensesnittet i applikasjonen.

Denne delen av dokumentasjonen er skrevet for en person med kompetanse innen programmering og systemutvikling. Dokumentasjonen er ment som et oppslagsverk for å gi en utenforstående innsikt i applikasjonens oppbygning og funksjonalitet. Ved hjelp av produktdokumentasjonen skal vedkommende kunne fortsette arbeidet med applikasjonen, både vedlikehold og videre utvikling. Dokumentasjonen er derfor meget teknisk.

Ord og uttrykk forklares fortløpende i fotnoter. I tillegg er samtlige fremmedord og forkortelser samlet i dokumentet "Ordforklaringer og kilder" bakerst i rapporten.

Innholdsfortegnelse

1 Datasett

Data i Kjørehjelperen hentes i hovedsak fra Nasjonal vegdatabank, NVDB, forvaltet av Statens vegvesen. I tillegg benyttes data fra MapQuest[1] til å bestemme avstanden til kommende objekter. I de følgende punktene tar vi for oss innholdet og strukturen i disse datasettene.

1.1 JSON

JavaScript Object Notation, kalt JSON, er en enkel tekstbasert standard for datautveksling. Den er opprinnelig avledet fra JavaScript for å representere enkle datastrukturer. Standarden er imidlertid uavhengig av JavaScript eller andre programmeringsspråk. (…) JSON-formatet brukes ofte til serialisering og overføring av strukturert data over en nettverkstilkobling, primært mellom en server og en webapplikasjon.[2]

JSON brukes ofte som et alternativ til XML[3]. Data kodet i JSON er generelt mindre enn tilsvarende data kodet i XML, blant annet på grunn av XML's avsluttende tagger. I tillegg mener mange at JSON parses[4] raskere enn XML.

```
{
  "firstName" : "John",
  "lastName" : "Smith",
  "age" : 25,
  "address": {
    "streetAddress" : "21 2nd Street",
    "city" : "New York",
    "state" : "NY",
    "postalCode" : "10021"
  },
  "phoneNumber" : [
    {
      "type" : "home",
      "number" : "212 555-1234"
    },
    {
      "type" : "fax",
      "number" : "646 555-4567"
    }
  ]
}
```

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>
      21 2nd Street
    </streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber type="home">
      212 555-1234
    </phoneNumber>
    <phoneNumber type="fax">
      646 555-4567
    </phoneNumber>
  </phoneNumbers>
</person>
```

Kodesnutt 1: Data formatert som JSON til venstre, og som XML til høyre. Hvis vi fjerner unødvendige mellomrom og linjeskift (brukt her for å gjøre koden lettleselig) består JSON-formatet av 240 tegn mens XML-formatet består av 344 tegn.

[1] MapQuest er en online karttjeneste som tilbyr blant annet vegkart og navigasjonshjelp.
[2] Hentet fra artikkelen om JSON på Wikipedia.org (http://no.wikipedia.org/wiki/JSON).
[3] Extensible Markup Language (XML) er et universelt og utvidbart markeringsspråk. Brukes til deling av strukturerte data mellom informasjonssystemer og til koding av dokumenter og som kommunikasjonsmiddel mellom ulike informasjonssystemer og dataformater.
[4] Prosessen med å generere eller lese data fra (i dette tilfellet) en JSON- eller XML-stuktur.

Både APIet[5] fra NVDB og APIet fra MapQuest tilbyr data både som JSON og XML. Vi har valgt å bruke JSON konsekvent, på grunn av fordelene knyttet til størrelse og hastighet. Det er i tillegg et krav at søkeobjekter[6] sendt til NVDB er på JSON-formatet, så det er derfor naturlig at det er samsvar mellom utgående og inngående objekter i applikasjonen.

1.2 Nasjonal vegdatabank

Nasjonal vegdatabank (NVDB) er en database med informasjon om riks- og fylkesveger, kommunale veger, private veger og skogsbilveger. Databasen brukes aktivt i Norges vegforvaltning. Den inneholder blant annet informasjon om vegnett med geometri og topologi som danner grunnlaget for kartløsninger og ruteberegnere på internett. I tillegg finnes en oversikt over utstyr og drenering langs vegen, ulykker og trafikkmengder og grunnlagsdata for bruk i støyberegning og trafikkmodellering.

Statens vegvesen

Figur 1: Dataene i NVDB eies og forvaltes av Statens vegvesen.

APIet er basert på REST[7], og kan brukes til å hente de fleste grunnlagsdata i NVDB. Dataene leveres på XML- eller JSON-format. I dag inneholder APIet mer enn 16 millioner objekter, fordelt på over 250 objekttyper.

1.2.1 Dataeier og lisens

Statens vegvesen eier og forvalter dataene i NVDB. Dataene er lisensiert under Norsk Lisens for Offentlige Data (NLOD)[8]. Denne lisensen er utarbeidet av Fornyings-, administrasjons- og kirkedepartementet og er ment brukt av offentlige virksomheter ved tilgjengeliggjøring av offentlige data. Lisensen gir kort oppsummert rett til fri bruk, også kommersielt, mot navngivelse av lisensgiver og at dataene ikke fremstilles villedende. I tilfellet med NVDB må applikasjonen inneholde teksten ”Inneholder data under norsk lisens for offentlige data (NLOD) tilgjengeliggjort av Statens vegvesen.”.

[5] Application Programming Interface (API) er et grensesnitt for kommunikasjon mellom programvare. APIet beskriver de metoder som en gitt programvare eller et bibliotek kan kommunisere med.
[6] Les mer om dette under "1.2.2.2 Finne objektene på den vegen vi befinner oss".
[7] Representational State Transfer, REST, er en programvarearkitektur for distribuerte systemer som World Wide Web. REST har etter hvert blitt den dominerende designmodellen for web-APIer.
[8] http://data.norge.no/nlod/no/1.0

1.2.2 Bruk av Nasjonal vegdatabank API

Når vi spør mot NVDB APIet bruker vi GET som innebærer at vi sender med informasjon til serveren i URL'en[9]. Vi sender i tillegg med en header[10] som forteller at vi ønsker å motta JSON-data. APIet har en grunn-URL som er følgende:

https://www.vegvesen.no/nvdb/api/

Ved spesifikke søk mot APIet legger vi til data på slutten av denne URLen.

1.2.2.1 Finne ut hvor brukeren befinner seg

NVDB APIet har funksjonalitet for å finne ut på hvilken veg brukeren befinner seg. Ved å sende koordinater til APIet returnerer det en *vegreferanse*, så sant det eksisterer en veg i nærheten av dette punktet. En vegreferanse inneholder informasjon om en veg, som for eksempel hva slags veg det er, i hvilken kommune den ligger osv. Viktigst for oss så inneholder den også en *veglenke*.

Følgende URL spør etter nærmeste veg til punktet 10.198832, 59.625669[11]:

…/vegreferanse?srid=WGS84&x=10.198832&y=59.625669

"srid=WGS84" angir at koordinatsystemet som brukes er WGS84, som er det systemet som brukes i lokasjonstjenesten i iOS. Spørringen over gir følgende svar fra NVDB:

```
{
  "sokePunkt" : "POINT (10.198832 59.625669)",
  "sokePunktSrid" : "WGS84",
  "punktPaVegReferanseLinjeUTM33" : "POINT (229343.56174456686 6619511.951859929)",
  "punktPaVegReferanseLinjeWGS84" : "POINT (10.198944775244298 59.62563354138125)",
  "meterVerdi" : 3702,
  "visningsNavn" : "EV 18 HP2 m3702",
  "veglenkeId" : 2172077,
  "veglenkePosisjon" : 0.8449232402200674,

  ...
}
```

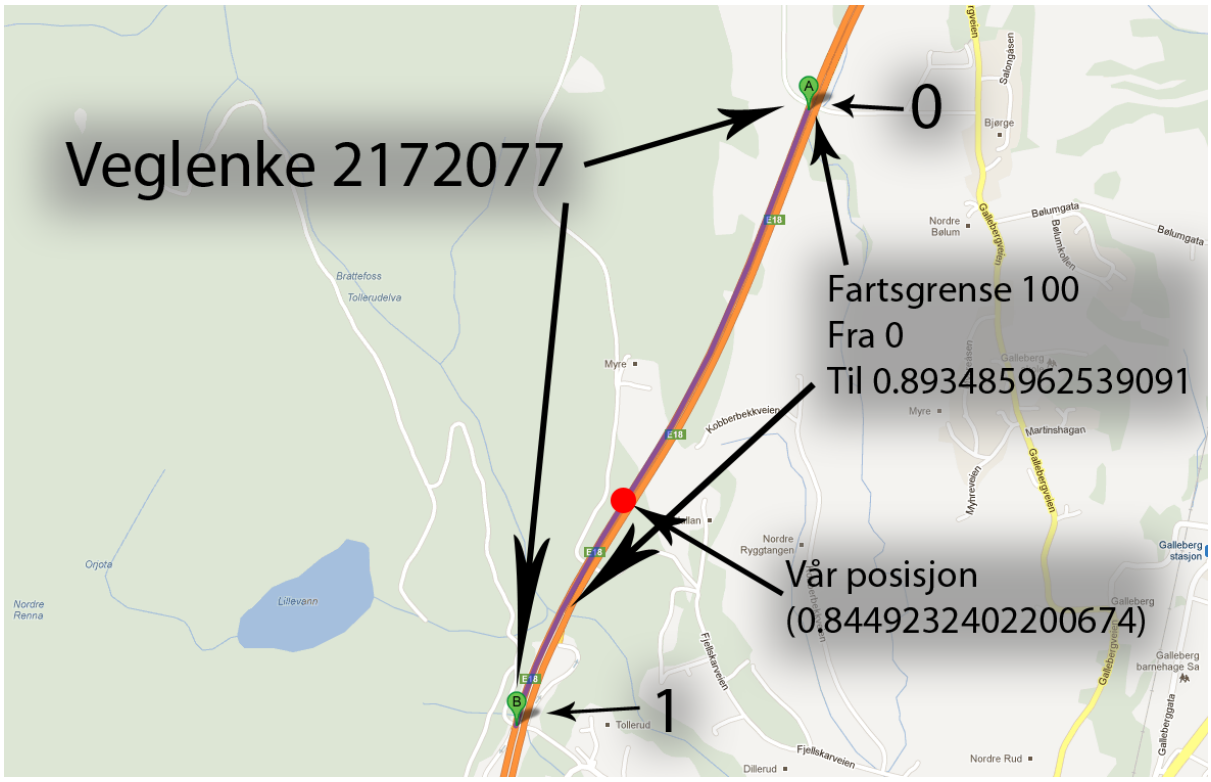
Kodesnutt 2: De første linjene av svaret fra NVDB når det spørres etter en vegreferanse på punkt 10.198832, 59.625669.

En veglenke er en sammenhengende vegstrekning med en unik ID. Posisjoner langs en veglenke defineres med et tall mellom 0 og 1. Vegreferansen som mottas når vi spør APIet om posisjonen vår forteller oss hvilken veglenke vi er på og hvilken posisjon vi befinner oss på på denne veglenken.

Hvis vi bruker eksempelet over ser vi at vi befinner oss på en veglenke med id 2172077 på posisjon 0.8449232402200674. Sett at vi senere finner en fartsgrense hvor det er oppgitt at den gjelder på

[9] Uniform Resource Locator.
[10] Supplerende data plassert først i en datablokk som blir lagret eller sendt.
[11] I Norge oppgis lengde- og breddegrader i motsatt rekkefølge av f.eks. USA. I Norge skriver vi "10.198832, 59.625669", mens i USA skriver de "59.625669, 10.198832".

veglenke 2172077 fra posisjon 0 til posisjon 0.893485962539091. Vi kan dermed vite at fartsgrensen gjelder der vi er, og trenger ikke forholde oss til avstander eller koordinater.



Figur 2: Bildet viser veglenken fra eksempelet. Vi ser at vår posisjon er mellom fra- og til-posisjonen til fartsgrensen med verdi 100, og kan dermed vite at den gjelder der vi er. (Kart: Google Maps)

1.2.2.2 Finne objektene på den vegen vi befinner oss

Når vi nå skal finne objekter på vegen vi er på, benytter vi veglenkeidentifikasjonen og –posisjonen vi mottok da vi spurte etter vegreferansen. Når vi søker etter objekter i NVDB må vi riktignok sende med et søkeobjekt på JSON-format.

```
{
  lokasjon:
  {
    veglenker:
    [
      {
        id: 2172077,
        fra: 0,
        til: 1
      }
    ]
  },
  objektTyper:
  [
    {
      id: 105,
      antall: 0,
      filter: []
    }
  ]
}
```

Kodesnutt 3: Et søkeobjekt på JSON-format, brukt til å søke etter objekter i NVDB.

Søkeobjektet i kodesnutt 3 søker etter fartsgrenser på vegen vi er på. Under lokasjon er det oppgitt en veglenke med id 2172077, med fra- og til-verdier på henholdsvis 0 og 1. Dette betyr at vi søker på hele veglenken med id 2172077. Under objekttyper søker vi etter et objekt med id 105. Dette er den numeriske id'en til fartsgrenser i NVDB. Når antall er satt til 0 betyr det at vi vil ha alle objektene den finner. Filteret står tomt her, men kan brukes til å filtrere hvilke objekter vi ønsker, eller ikke ønsker. I denne sammenhengen kunne vi for eksempel bedt om å kun motta fartsgrenser som har verdien 50.

Satt inn i URL'en ser spørringen mot NVDB slik ut:

…/sok?kriterie={lokasjon:{veglenker:[{id:2172077,fra:0,til:1}]},objektTyper:[{id:105,antall:0,filter:[]}]}

Med denne spørringen får vi følgende svar fra NVDB:

```
{
  "sokeObjekt": {...},
  "totaltAntallReturnert": 4,
  "resultater":
  [
    {
      "typeId": 105,
      "statistikk": {...},
      "vegObjekter":
      [
        {
          "objektId": 87557714,
          "objektTypeId": 105,
          "objektTypeNavn": "Fartsgrense",

          ...

          "egenskaper":
```

```
[
  {...},
  {
    "id": 2021,
    "navn": "Fartsgrense",
    "verdi": "100",
    "enumVerdi": {...},
    "definisjon": {...}
  },

  ...

],
strekningslengde: 8630,
lokasjon:
{

...

veglenker:
[
  {
    "id": 2172077,
    "fra": 0,
    "til": 0.893485962539091,
    "direction": "WITH"
  }
],

...
```

Kodesnutt 4: Svar fra NVDB på JSON-format. Her har vi spurt etter fartsgrenser på veglenke 2172077.

Svaret vi mottok fra NVDB over inneholder 4 fartsgrenser. Det som gjenstår for å finne ut hvilken fartsgrense som gjelder akkurat der vi er vil derfor være å gå gjennom fartsgrensene og se om en av dem dekker det punktet på veglenken vi befinner oss på.

Vi finner en fartsgrense som gjelder på veglenke 2172077 fra 0 til 0.893485962539091. Som vi vet fra eksempelet over befinner vi oss på den samme veglenken, og på posisjon 0.8449232402200674. Vi vet dermed at dette er gjeldende fartsgrense-objekt der vi er. Under egenskaper i svaret ser vi at fartsgrensen er 100.

1.2.2.3 Utvidelser og praktisk bruk

Måten vi søker etter en vegreferanse i eksempelet over er det samme som gjøres i Kjørehjelperen. Vi gjør denne spørringen hver gang vi mottar nye koordinater fra posisjonstjenesten i telefonen. Dette er nødvendig, siden vi trenger hjelp av NVDB til å avgjøre hvilken veg vi befinner oss på til enhver tid.

Når vi søker etter objekter langs en veg i eksempelet over, sender vi kun med én objekttype, altså fartsgrense. "Objekttyper" i søkeobjektet er et array[12], noe som betyr at vi kan sende med flere objekttyper. Det er dette vi gjør i applikasjonen. Vi søker etter alle 40 objekttypene samtidig, uavhengig av om brukeren ønsker å bli varslet om disse eller ikke. Alle objektene lagres deretter på

[12] Et array er en endimensjonal tabell.

telefonen, slik at neste gang brukeren befinner seg på vegen med denne id'en, lastes dataene fra telefonen i stedet.

1.3 MapQuest

MapQuest er en online karttjeneste som tilbyr blant annet vegkart og navigasjonshjelp. De leverer tjenester til bruk med blant annet JavaScript[13], Flash[14] og webtjenester, i tillegg til egne kartapplikasjoner til forskjellige smarttelefoner. MapQuest eies av amerikanske AOL[15].

Figur 3: Avstandene til objekter i applikasjonen benytter MapQuest Open Directions API Web Service.

I Kjørehjelperen benyttes MapQuest Open Directions API Web Service. Denne tjenesten benytter OpenStreetMap[16] til å levere navigasjonsdata direkte over HTTP[17].

1.3.1 Valg av MapQuest

NVDB inneholder ingen avstander eller lengder. Vi var derfor nødt til å benytte en annen tjeneste for å beregne avstanden frem til punkter av interesse. Google Maps ble også vurdert som mulig leverandør av navigasjonstjeneste, men ble forkastet grunnet lisensieringsproblemer[18]. Valget falt på MapQuest på grunn av den åpne lisensen.

1.3.2 Bruk av tjenesten

På samme måte som NVDB APIet sendes spørringen til MapQuest APIet som en URL med parametere. Vi bygger også her et søkeobjekt som JSON:

```
{
  locations:
  [
    {
      latLng:
      {
        lat:59.652683,
        lng:10.226554
      }
    },
    {
```

[13] JavaScript er et skriptspråk som er best kjent for å tilføre dynamiske elementer til nettsider.
[14] Adobe Flash er programvare som utvikles og distribueres av Adobe Systems. Flash brukes mest til å vise dynamisk innhold på nettsider og støtter vektor- og pikselgrafikk.
[15] AOL Inc. (Tidligere kjent som America Online).
[16] OpenStreetMap (OSM) er et fritt redigerbart kart over hele jorden, laget og redigert av brukerne. OpenStreetMap er lisensiert under Open Data Commons Open Database License (ODbL), se http://opendatacommons.org/licenses/odbl/1.0/ for mer info.
[17] Hypertext Transfer Protocol.
[18] Google krever at deres navigasjonstjeneste benyttes i deres eget kart.

```
      latLng:
      {
        lat:59.633401,
        lng:10.205687
      }
    }
  ],
  options:
  {
    unit:k
  }
}
```

Kodesnutt 5: Et søkeobjekt på JSON-format som sendes til MapQuest APIet.

Som man ser av eksempelet over inneholder søkeobjektet to koordinatpunkter, det første er vår posisjon, mens det andre er posisjonen til objektet vi ønsker å finne avstanden til. Under "options" settes "unit" til "k". Dette sørger for at vi mottar et svar i kilometer. Vi legger søkeobjektet inn i parameteren "json" og får følgende URL som sendes til MapQuest:

http://open.mapquestapi.com/directions/v1/routematrix?key=X&json={locations:[{latLng:{lat:59.652683, lng:10.226554}},{latLng:{lat:59.633401,lng:10.205687}}],options:{unit:k}}

Legg merke til den ekstra parameteren "key=X" i URLen over. For å bruke APIet kreves det at man registrerer applikasjonen hos MapQuest. Kjørehjelperen har en lang og unik nøkkel som sendes med i spørringen i stedet for X.

Spørringen over gir følgende svar fra MapQuest:

```
{
  "allToAll" : false,
  "distance" :
  [
    0,
    2.465
  ],

  ...
```

Kodesnutt 6: Et eksempel på svar fra MapQuest.

Svaret inneholder en del mer informasjon enn det som kommer frem av eksempelet over. Vi er allikevel kun interesserte i én ting: Det andre tallet under "distance". Dette tallet representerer avstanden mellom de to punktene vi oppga. Det er altså 2,465 km fra vår posisjon til objektet foran oss.

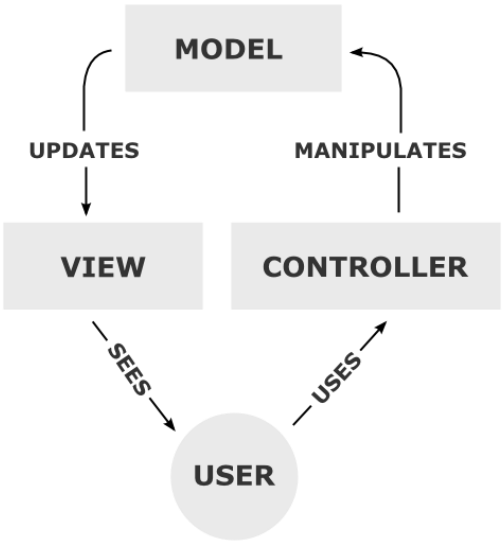
2 Design pattern

I dette kapittelet tar vi for oss MVC, hvordan det er implementert i iOS[19] og hvordan vi bruker det i Kjørehjelperen.

2.1 Model View Controller

Model View Controller, eller MVC, er et programvarearkitektur-pattern[20] som separerer representasjonen av data fra hvordan det fremstilles for brukeren.

Modellen består av selve datastrukturen, i tillegg til regler, logikk og funksjoner. I objektorientert programmering faller selve objektene inn under modellen. I tillegg får funksjoner som interagerer med databaser eller andre eksterne tjenester plass her.



Figur 4: Modellen, viewet og kontrolleren i forhold til brukeren (Foto: Wikimedia Commons).

Viewet er det brukeren ser, altså brukergrensesnittet. Frontend-kode, som HTML[21], Javas[22] Swing-bibliotek eller iOS's UIView-klasse er eksempler på kode som kun har noe med hvordan brukeren ser informasjon.

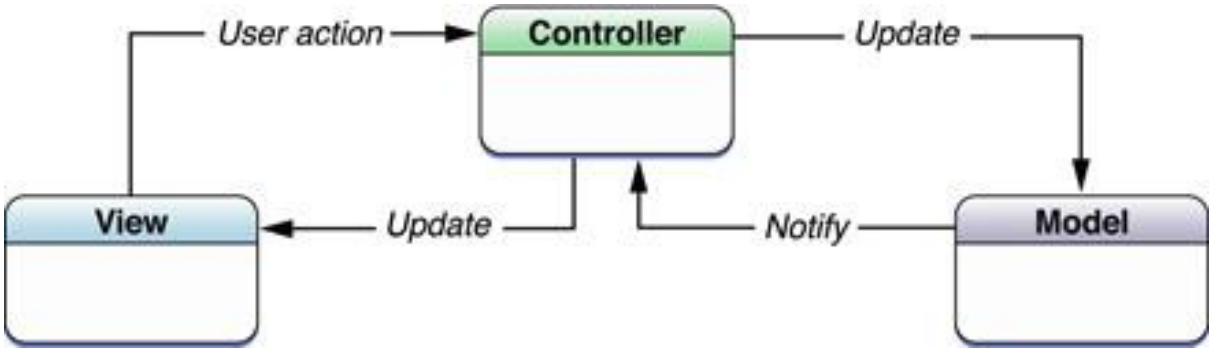
Kontrolleren (Controller) er bindeleddet mellom modellen og viewet. Kontrollerens oppgave er å formidle informasjonen fra modellen til viewet, og å sende eventuelle endringer fra brukeren tilbake til modellen.

De sentrale ideene bak MVC er gjenbruk av kode og separering av ansvar. Modellen trenger ikke tenke på hvordan dataene fremstilles for brukeren, og viewet trenger ikke bry seg om hvor dataene det viser kommer fra. Det er kontrollerens oppgave å binde de to andre sammen. Tanken er dermed at man kan bytte ut enkeltdeler av koden ved bytte av f.eks. datagrunnlag eller plattform, uten å måtte skrive all kode på nytt.

[19] iOS er Apples operativsystem for iPhone, iPad og iPod.
[20] Et designpattern er en kjent gjenbrukbar løsning på et vanlig problem i systemutvikling.
[21] HyperText Markup Language (HTML) er et markeringsspråk for formatering av nettsider.
[22] Java er et objektorientert programmeringsspråk utviklet av Sun Microsystems (nå kjøpt av Oracle Corporation).

2.2 MVC i iOS

iOS følger også MVC-patternet. Det har modeller, views og kontrollere. Det spesielle er at i en iOS-applikasjon er gjerne viewet slått sammen med kontrolleren i en såkalt ViewController. En ViewController er en kontroller som fokuserer mer på viewet enn modellen. Den "eier" et eller flere views, og hovedoppgaven dens er å administrere brukergrensesnittet. Den er riktignok fremdeles to separate elementer, en kontroller med et tilhørende view. Grunnen til denne fremgangsmåten er at i en iOS-applikasjon oppdateres ofte viewet med en gang det skjer en endring i modellen. Slettes det f.eks. en innføring i en tabell må modellen oppdateres, og i tillegg forventes det at raden forsvinner fra viewet med en gang. Det er i tillegg ønskelig at raden forsvinner uten at hele viewet må tegnes på nytt.



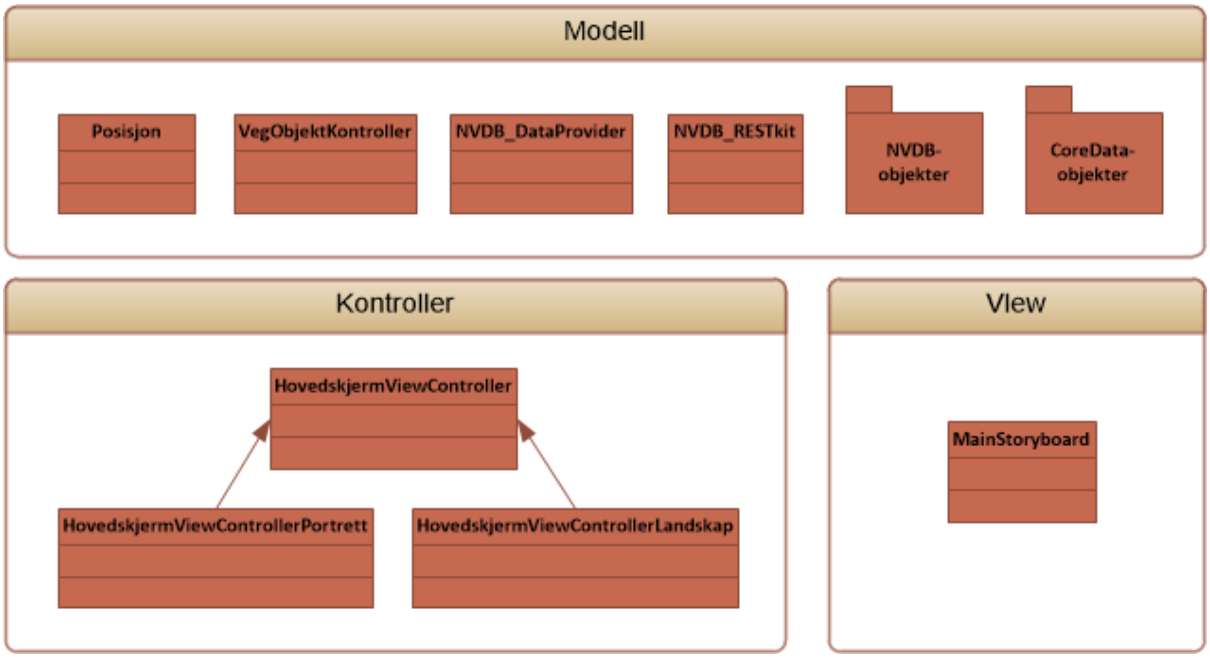
Figur 5: Denne figuren viser hvordan Apple beskriver forholdet mellom modellen, viewet og kontrolleren (Foto: Apple).

2.3 MVC i Kjørehjelperen

Kjørehjelperen bruker naturlig nok også MVC-patternet. Figur 22 gir en forenklet oversikt over hvor de ulike klassene i applikasjonen passer inn i de tre MVC-lagene. For å gi en forståelse av hvorfor de forskjellige klassene er plassert der de er, gis det også en kort beskrivelse av de ulike klassenes funksjon og oppgave.

Modellaget er det absolutt største laget. Her har vi Posisjon, en klasse som finner GPS-posisjonen til telefonen. VegObjektKontroller inneholder logikken som bestemmer hva som skal vises på skjermen til brukeren. NVDB_DataProvider sørger for å levere data til VegObjektKontroller, enten fra NVDB eller fra Core Data[23]. NVDB_RESTkit sørger for kommunikasjon med NVDB APIet og MapQuest APIet. NVDB-objekter og CoreData-objekter er objekter som representerer konkrete objekter, f.eks. en fartsgrense eller en jernbaneovergang.

[23] Core Data er et iOS-rammeverk som lagrer data i en database lokalt på enheten. Les mer under "3.1 Core Data".



Figur 6: De ulike klassene i Kjørehjelperen tilhører enten modell-, kontroll- eller view-laget i MVC.

Kontrollaget inneholder tre klasser, hvorav HovedskjermViewControllerPortrett og HovedskjermViewControllerLandskap er subklasser[24] av HovedskjermViewController. Subklassene gjør bare ting som er spesifikt for orienteringen av telefonen, det meste av logikken ligger i HovedskjermViewController. Denne klassen er ansvarlig for å vise informasjonen den mottar fra VegObjektKontroller på skjermen. I tillegg utfører den oppgaver knyttet til oppstart og avslutning av applikasjonen, endring av innstillinger og endring av orientering.

Viewet inneholder kun en klasse, MainStoryboard. Denne klassen, eller storyboardet[25], beskriver hvordan skjermbildet er bygget opp ved hjelp av XML. Storyboardet lages med et grafisk verktøy i Xcode[26], som så genererer XML-koden automatisk.

En mer detaljert beskrivelse av de forskjellige klassene og interaksjonen mellom dem er forklart under kapittel 4.2, Klassene i Kjørehjelperen.

[24] En subklasse er en klasse som arver alle egenskapene til en superklasse, eller en foreldreklasse.
[25] Et storyboard er et dokument som beskriver layoutet til et skjermbilde og overgangene mellom dem i iOS.
[26] Xcode er et program til Mac for utvikling, testing og publisering av programmer til iOS og OS X.

3 Sentrale rammeverk

Når man utvikler applikasjoner til iOS eller andre mobile plattformer er det mye som skjer i bakgrunnen som man slipper å tenke på som utvikler. Takket være solide SDKer[27] får man tilgang til posisjon, kontaktlister og kamera for å nevne noe, med enkle funksjonskall. Skulle vi gått i detalj på alle rammeverkene vi har benyttet oss av i iOS ville dette dokumentet blitt alt for omfattende. Vi har derfor valgt å gå inn på to sentrale rammeverk som har spart oss for mye arbeid.Det ene er Core Data, et rammeverk i iOS som lagrer data lokalt på enheten. Det andre er RestKit, et åpent rammeverk som forenkler kommunikasjonen med REST-baserte webtjenester og mappingen[28] av data til objekter.

3.1 Core Data

Core Data is an object graph and persistence framework provided by Apple.[29] Med dette menes det at Core Data lagrer instanser av klasser på et gitt tidspunkt, og forholdet mellom dem. Det forteller også at Core Data fungerer som et abstrakt lag mellom en applikasjon og de lagrede dataene, for eksempel en database.

Figur 7: Ikonet til Core Data (Foto: Apple).

For å si det på en enklere måte: Core Data lar deg opprette objekter i applikasjonen på vanlig måte. Det er så Core Datas oppgave å lagre disse objektene, oppdatere eventuelle endringer som gjøres, og å hente dem inn igjen neste gang applikasjonen startes. Som utvikler trenger man ikke tenke på hvor eller hvordan disse lagres.

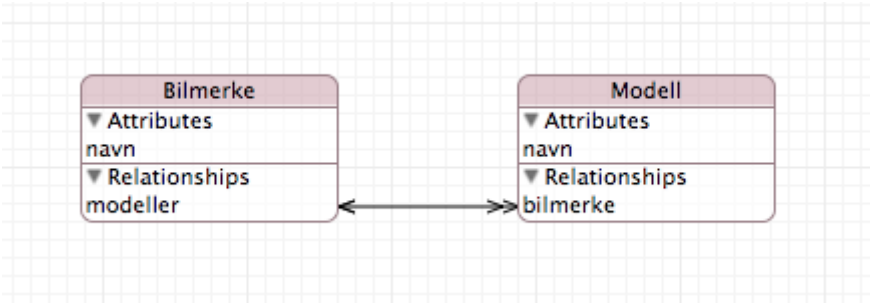
Core Data bruker gjerne en SQLite-database[30] til å lagre data, men man trenger altså ikke skrive SQL[31] for å benytte seg av dette. Det er allikevel noe arbeid som må gjøres for å komme i gang med Core Data.

Figur 8: Core Data bruker gjerne en SQLite-database (Foto: Wikimedia Commons).

For å benytte Core Data i et iOS-prosjekt må man tegne opp alle entitetene[32] som ønskes og forholdet

[27] Software development kit (SDK) er verktøy som lar deg utvikle programmer til en spesifikk programvarepakke, et rammeverk, en hardware-plattform, et operativsystem e.l.
[28] Mapping er prosessen med å overføre data fra en datastruktur til en annen, f.eks. fra JSON til et NSObject (objekt i Objective-C).
[29] Hentet fra artikkelen om Core Data på Wikipedia.org (http://en.wikipedia.org/wiki/Core_Data).
[30] SQLite er et relasjonsdatabasesystem i et lite C-bibliotek. I motsetning til andre databasesystemer er SQLite integrert i klientapplikasjonen og ikke en separat prosess.
[31] Structured Query Language (SQL) er et programmeringsspråk brukt til å kommunisere med relasjonsdatabaser.
[32] En entitet kan defineres som noe som er i stand til selvstendig eksistens og som kan identifiseres.

mellom dem. Dette gjøres med et grafisk verktøy i Xcode. Man må gi entitetene navn, attributter og relasjoner. For å ta et klassisk eksempel: Du oppretter en entitet som heter "Bilmerke" med et attributt kalt "navn". Deretter oppretter du en ny entitet kalt "Modell" med et annet attributt også kalt "navn". "Bilmerke" får så en en-til-mange-relasjon til "Modell" kalt "modeller" siden hvert bilmerke som oftest har mange modeller, mens hver modell kun har ett "Bilmerke".



Figur 9: Entitetene "Bilmerke" og "Modell" i Xcode.

Xcode genererer deretter klasser for deg basert på disse entitetene.

```
// Bilmerke.h

#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@class Modell;

@interface Bilmerke : NSManagedObject

@property (nonatomic, retain) NSString * navn;
@property (nonatomic, retain) NSSet * modeller;
@end

@interface Bilmerke (CoreDataGeneratedAccessors)

- (void)addModellerObject:(Modell *)value;
- (void)removeModellerObject:(Modell *)value;
- (void)addModeller:(NSSet *)values;
- (void)removeModeller:(NSSet *)values;

@end


// Bilmerke.m

#import "Bilmerke.h"
#import "Modell.h"

@implementation Bilmerke

@dynamic navn;
@dynamic modeller;

@end
```

Kodesnutt 7: Autogenerert klasse i Xcode basert på entiteten "Bilmerke". En klasse i Objective-C består av to filer, en deklarasjonsfil (*.h) og en implementasjonsfil (*.m). Les mer om Objective-C under 4.1.1 Objective-C.

I kodesnutt 7 ser vi den autogenererte klassen Bilmerke. Klassen har fått en variabel "navn" og et sett[33] av modeller. I tillegg har Core Data generert fire metoder for å legge til eller fjerne en eller flere modeller.

Man kan nå opprette objekter i applikasjonen på samme måte som man ellers ville gjort. Man må riktignok kalle en spesiell metode ved opprettelse av et nytt objekt, og man må huske å fortelle Core Data at man vil lagre endringene i det nye objektet.

```
Bilmerke * nyttMerke = [NSEntityDescription
                                    insertNewObjectForEntityForName:@"Bilmerke"
                                    inManagedObjectContext:managedObjectContext];
[nyttMerke setNavn:@"Toyota"];

Modell * nyModell = [NSEntityDescription
                                    insertNewObjectForEntityForName:@"Modell"
                                    inManagedObjectContext:managedObjectContext];
[nyModell setNavn:@"Avensis"];

[nyttMerke addModellerObject:nyModell];

NSError * error;
[managedObjectContext save:&error];
```

Kodesnutt 8: Eksempel på opprettelse av et bilmerke "Toyota" med en modell "Avensis".

I den første linjen i kodesnutt 8 oppretter vi et Bilmerke-objekt. "insertNewObjectForEntityName:@"Bilmerke"" betyr at vi vil knytte Bilmerke-objektet til entiteten "Bilmerke" i Core Data. "inManagedObjectContext:managedObjectContext" betyr at Core Data-laget vi vil bruke er definert i variabelen managedObjectContext. Denne variabelen ble satt da applikasjonen startet. Deretter setter vi navnet på merket, og oppretter et Modell-objekt på samme måte. Modell-objektet legger vi deretter til merket med "addModellerObject:nyModell". Til slutt må vi fortelle Core Data at vi ønsker å lagre de nye objektene. Dette gjøres gjennom managedObjectContext. Vi må sende med en referanse til NSError når vi lagrer for å plukke opp eventuelle feil som skulle oppstå.

Når vi ønsker å hente ut objekter fra Core Data må vi opprette en NSEntityDescription. Denne legges inn i et NSFetchRequest som kjøres av managedObjectContext. Dette returnerer et array bestående av alle entitetene av denne typen i Core Data.

```
NSEntityDescription * bilmerkeEntity = [NSEntityDescription
                                    entityForName:@"Bilmerke"
                                    inManagedObjectContext:managedObjectContext];
NSFetchRequest * request = [[NSFetchRequest alloc] init];
[request setEntity:bilmerkeEntity];
```

[33] Et sett (NSSet i Objective-C) er en datastruktur som holder på mange elementer. Elementene er ikke knyttet til en fast plass slik som i et array.

```
NSError * error;
NSArray * resultat = [managedObjectContext executeFetchRequest:request
                                                          error:&error];
```

Kodesnutt 9: Eksempel på uthenting av bilmerker fra Core Data.

Variabelen "resultat" peker nå på et array som inneholder alle bilmerkene i Core Data. Innholdet i "Bilmerke"-entiteten er gjort om til Bilmerke-objekter. På grunn av relasjonene vi oppga da vi designet entitetene inneholder også Bilmerke-objektene sine tilhørende Modell-objekter.

3.2 RestKit

RestKit er et Objective-C[34]-rammeverk for iOS som forenkler kommunikasjonen med REST-baserte webtjenester og mappingen av objekter. Rammeverket fungerer slik at man oppgir URLen til APIet, forteller hvilke objekter man forventer å motta, og beskriver hvordan de mottatte dataene skal mappes til disse objektene. Resten ordner RestKit. Man kjører et asynkront metodekall til RestKit, og deretter leverer RestKit de ferdig mappede objektene når ting er ferdig.

RestKit er veldig kraftig, og har mange funksjoner, blant annet støtte for Core Data. Eksemplene i de påfølgende kodesnuttene er gjort så enkle som mulig for å forstå den grunnleggende funksjonaliteten i RestKit.

```
{
  personer :
  [
    {
      fornavn : "Lars"
      etternavn : "Smeby"
    },
    {
      fornavn : "Henrik"
      etternavn : "Hermansen"
    }
  ]
}
```

Kodesnutt 10: Et JSON-objekt bestående av to personer.

I kodesnutt 10 ser vi et eksempel på hva vi forventer å motta fra APIet. Her har vi to person-objekter, hver med et fornavn og et etternavn.

```
// Person.h

#import <Foundation/Foundation.h>

@interface Person : NSObject

@property (nonatomic, strong) NSString * fornavn;
@property (nonatomic, strong) NSString * etternavn;
```

[34] Objective-C er programmeringsspråket som brukes av Apple. Det er basert på C.

```
@end
```

Kodesnutt 11: Deklarasjonen av Person-objektet vårt i Xcode.

I Xcode deklarerer vi så et Person-objekt med fornavn og etternavn. Kodesnutt 11 viser deklarasjonsfilen (headerfilen[35]) til Person-objektet.

```
NSURL * grunnURL = [NSURL URLWithString:@"http://eksempel.no/api"];
AFHTTPClient * klient = [AFHTTPClient clientWithBaseURL:grunnURL];
[klient setDefaultHeader:@"Accept" value:RKMIMETypeJSON];

RKObjectMapping * personMapping = [RKObjectMapping
                                              mappingForClass:[Person class]];
[personMapping addAttributeMappingsFromDictionary:@{@"fornavn" : @"fornavn",
                                                    @"etternavn" : @"etternavn"}];

RKResponseDescriptor * responseDescriptor = [RKResponseDescriptor
                                    responseDescriptorWithMapping:personMapping
                                                         pathPattern:nil
                                                             keyPath:@"personer"
          statusCodes:RKStatusCodeIndexSetForClass(RKStatusCodeClassSuccessful)];

RKObjectManager * objectManager = [[RKObjectManager alloc]
                                                    initWithHTTPClient:klient];
[objectManager addResponseDescriptor:responseDescriptor];

[objectManager getObjectsAtPath:@"http://eksempel.no/api/personer" parameters:nil
   success:^(RKObjectRequestOperation *operation, RKMappingResult *mappingResult)
   {
      // Hvis alt går bra kommer vi hit. Resultatet ligger i parameteren
      // mappingResult.
   }
   failure:^(RKObjectRequestOperation *operation, NSError *error)
   {
      // Hvis det har skjedd en feil kommer vi hit. Beskrivelse av feilen ligger i
      // parameteren error.
   }];
```

Kodesnutt 12: Eksempel på bruk av RestKit for å hente ned JSON-data og mappe det til Person-objekter.

Kodesnutt 12 viser bruken av RestKit for å hente ned JSON-data og mappe det til Person-objekter. Eksempelet inneholder mange statiske[36] metodekall til RestKit-klasser som vi ikke skal gå inn på. Vi tar kun for oss den overordnede gangen i eksempelet.

Først oppretter vi en klient med URLen til APIet og sier at vi ønsker å motta JSON-data. Deretter oppretter vi en mappingklasse. Her forteller vi at mottatte data skal mappes til Person-objekter og at "fornavn" i JSON-dataen skal mappes til "fornavn" i Person-objektet, og tilsvarende med "etternavn". Vi oppretter så en ResponseDescriptor som bestemmer hva som skal gjøres med resultatet av spørringen mot APIet. Med parameterne sier vi at vi skal bruke det nye mappingobjektet vi opprettet til å mappe det vi finner under "personer" i JSON-svaret. Til slutt oppretter vi en ObjectManager. Vi legger ResponseDescriptoren til ObjectManageren og sender

[35] En headerfil er en fil som forteller hvilke egenskaper og metoder en klasse har, men ikke hvordan de er implementert. Headerfiler benyttes bl.a. i C, C++ og Objective-C.
[36] Statiske metoder er metoder som ikke er knyttet til en spesifikk instans av en klasse.

avgårde en forespørsel. Hvis alt går bra havner vi i "success"-delen etter kallet med alle Person-objektene liggende i parameteren "mappingResult".

4 Programmets oppbygging og virkemåte

I dette kapittelet tar vi for oss hvordan Kjørehjelperen er bygget opp. Først tar vi for oss noen sentrale elementer i iOS og Objective-C som man må kjenne til. Deretter tar vi for oss hvilke klasser applikasjonen består av og deres funksjoner. Vi går ikke inn på hver enkelt metode, dette hadde blitt for omfattende. Applikasjonen inneholder nesten 13.000 kodelinjer, selv om mye av dette er autogenerert av Xcode. Til slutt følger vi gangen i applikasjonen fra den spør etter posisjonen til den viser vegskilt på skjermen.

4.1 iOS og Objective-C

Når man utvikler til iOS eller OS X[37] benytter man Objective-C. Vi skal se nærmere på syntaksen i dette språket og sammenlikne det med Java. Vi skal også se på iOS-spesifikke elementer som protokoller, delegater og storyboards.

4.1.1 Objective-C

Objective-C er et objektorientert høynivåspråk[38]. Det er basert på Smalltalk[39] og C. Objective-C er hovedprogrammeringsspråket brukt av Apple i OS X og iOS. Språket ligner på C++, C# og Java, men har en litt annerledes syntaks på metodekall.

```
- (int)adderTallA:(int)a MedB:(int)b
{
   return a + b;
}

...

int c = [adderTallA:4 MedB:5];
```

```
public int adder(int a, int b)
{
   return a + b;
}

...

int c = adder(4, 5);
```

Kodesnutt 13: En metodedeklarasjon og et kall på metoden i Objective-C (til venstre) og Java (til høyre).

På samme måte som i C og C++ deler Objective-C klasser opp i to filer, en deklarasjonsfil (headerfil) og en implementasjonsfil. Grunnen til dette er at når man inkluderer klassen i andre filer trenger man ikke vite noe om logikken i klassen, kun navnet på egenskaper og metoder. Det holder derfor å inkludere klassens headerfil (som gjerne er mye mindre enn implementasjonsfilen).

```
// Person.h

#import <Foundation/Foundation.h>

@interface Person : NSObject
```

[37] OS X er det gjeldende operativsystemet til Mac.
[38] Et høynivåspråk har stor abstraksjon fra måten datamaskinen fungerer.
[39] Smalltalk er et objektorientert programmeringsspråk. Det spesielle med Smalltalk er at sjekking av typer og objekter skjer under kjøring i motsetning til under kompilering.

```
@property (nonatomic, strong) NSString * fornavn;
@property (nonatomic, strong) NSString * etternavn;
- (NSString *)hentFulltNavn;

@end

// Person.m

#import "Person.h"

@implementation Person

@syntesize fornavn, etternavn;

- (NSString *)hentFulltNavn
{
   return [@{fornavn, etternavn} componentsJoinedByString:@" "];
}

@end
```

Kodesnutt 14: Header- og implementasjonsfilen til objektet Person i Objective-C.

I Objective-C deklarerer man objektet som et interface (som ikke må forveksles med et interface i Java) i headerfilen. Man angir deretter egenskapene med "@property". Nøkkelordet "nonatomic" betyr litt forenklet at egenskapen kan aksesseres av flere tråder[40]. "strong" har erstattet "retain" i ARC[41], og kreves når egenskapen er en peker[42] til et objekt. Nøkkelordet betyr at programmet skal holde rede på hvor mange objekter som er opprettet og slette dem fra minnet når de ikke brukes lenger. I interfacet deklarerer man også eventuelle metoder som skal være synlige utenfor klassen.

I implementasjonsfilen benyttes "@implementation" foran navnet på klassen. "@syntesize" gjør så variablene kan aksesseres direkte, uten å måtte bruke klassenavnet foran variabelnavnet.[43] Metodene deklarert i headerfilen må implementeres.

```
public class Person
{
   private String fornavn, etternavn;

   public String getFornavn {return fornavn;}
   public String getEtternavn {return etternavn;}
   public void setFornavn(String fornavn){this.fornavn = fornavn;}
   public void setEtternavn(String etternavn){this.etternavn = etternavn;}

   public String hentFulltNavn
   {
      return fornavn + " " + etternavn;
   }
}
```

Kodesnutt 15: Den samme klassen, Person, her skrevet i Java.

[40] En tråd er en sekvens av programinstrukser som kan kjøres uavhengig av andre instrukser i et program (en lettvekts-prosess).
[41] Automatic Reference Counting (ARC) er at ansvaret for minnehåndtering flyttes fra programmereren til kompilatoren.
[42] En peker er en referanse til et sted i minnet hvor objektet ligger.
[43] I tidligere versjoner av iOS måtte man dessuten bruke "@syntesize" for at variablene skulle være tilgjengelige utenfor klassen, tilsvarende å skrive get- og set-metoder i Java.

4.1.2 Cocoa Touch

Når man utvikler applikasjoner til iPhone benyttes Cocoa Touch, et brukergrensesnitt-rammeverk for iOS. Det bygger på Cocoa-rammeverket brukt i utvikling til OS X, og er et abstrakt lag som gir tilgang til hardware-egenskaper på enheten.

4.1.3 Protokoller

En protokoll i Objective-C kan minne om et interface i Java. Det er en kontrakt som spesifiserer hvilke metoder en klasse må implementere. Med protokoller kan du dermed "standardisere" klasser og vite at en klasse som implementerer en protokoll også har implementert protokollens metoder.

Det er riktignok mulig å ha valgfrie metoder i en protokoll med nøkkelordet "@optional". Hvis man velger å benytte dette bør man teste om en klasse har implementert denne metoden før man eventuelt kaller den.

Selv om det er mulig å bruke en protokoll på samme måte som et interface i Java, brukes det som oftest til å deklarere delegater.

```
@protocol EksempelProtokoll
@required
- (void) eksempelMetodeA:(NSString *)enStreng;
- (void) eksempelMetodeB:(NSNumber *)etTall;
@end

...

@interface EnKlasse : NSObject <EksempelProtokoll>

@end
```

Kodesnutt 16: Deklarasjonen av en protokoll og deklarasjonen av en klasse som implementerer denne protokollen. Denne klassen må implementere protokollens metoder i implementasjonsfilen for at programmet skal kompilere.

4.1.4 Delegater

Delegater er en viktig del av iOS. Kommunikasjonen mellom biblioteker og klasser er basert på delegater. Det muliggjør asynkrone kall til klasser uten å låse hele systemet mens man venter på et resultat.

I C og C++ har man funksjonspekere[44] som kan minne noe om delegater. Det finnes ikke noe direkte tilsvarende i f.eks. Java. Man kan riktignok få til noe liknende i Java ved å lage sin egen event[45] og lyttere. Når eventen kjøres vil alle klasser som lytter etter denne eventen kjøre en navngitt metode. Til sammenlikning vil en delegat kun kjøre en navngitt metode i en spesifikk instans av en klasse.

[44] En funksjonspeker er en peker som peker til kjørbar kode i minnet.
[45] En hendelse i et program. Klasser og metoder kan "lytte" etter hendelser.

La oss klargjøre med et eksempel fra Kjørehjelperen: Vi har deklarert en protokoll i Posisjon-klassen som heter PosisjonDelegate. Denne protokollen krever at klasser som implementerer den også implementerer en metode "- (void) posisjonOppdatering(Posisjon *)posisjon". I Posisjon-klassen har vi også en peker "delegate" som peker til en klasse som implementerer denne protokollen. Posisjon-klassen vet ingenting om klassen denne peker til.

Når systemet kommer frem til en ny posisjon kalles denne metoden i Posisjon-klassen. Klassen som implementerer denne protokollen kjører dermed denne metoden.

```
// Posisjon.h

@protocol PosisjonDelegate
@required
- (void) posisjonOppdatering:(Posisjon *)posisjon;
@end

...
```

Kodesnutt 17: PosisjonDelegate deklareres i Posisjon.h.

```
// Posisjon.m

...

if([self.delegate conformsToProtocol:@protocol(PosisjonDelegate)])
{
   Posisjon * posisjon = [Posisjon alloc];

   ...

   [self.delegate posisjonOppdatering:posisjon];
}

...
```

Kodesnutt 18: I Posisjon.m sjekkes det at "delegat"-variabelen implementerer PosisjonDelegate-protokollen før "posisjonOppdatering"-metoden kalles med en ny posisjon.

```
// EksempelKlasse.h

@interface EksempelKlasse : NSObject <PosisjonDelegate>

...

@end
```

Kodesnutt 19: EksempelKlasse.h implementerer PosisjonDelegate-protokollen.

```
// EksempelKlasse.m

@implementation EksempelKlasse

...

   Posisjon * pos = [[Posisjon alloc] init];
   pos.delegate = self;

...
```

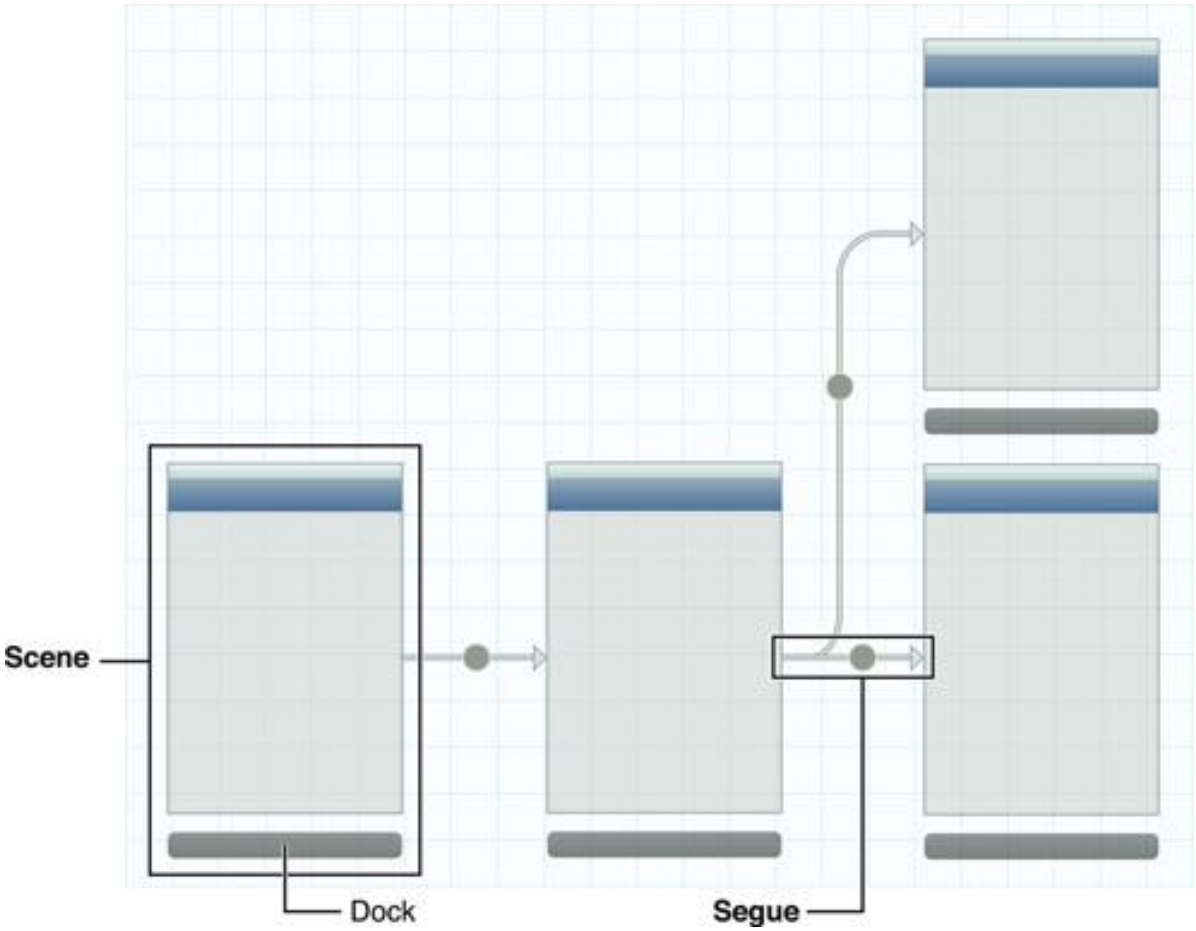
```
- (void) posisjonOppdatering(Posisjon *)posisjon
{
   // Gjør noe med posisjonen.
}

@end
```

Kodesnutt 20: I EksempelKLasse-klassen opprettes en instans av Posisjon-klassen. Her settes Posisjon-klassens "delegate"-variabel til instansen av EksempelKlasse-klassen. "posisjonOppdatering"-metoden er også implementert. Denne kjøres dermed når den kalles fra Posisjon-klassen.

Den største fordelen med delegater er at man kan separere ansvarsområder i applikasjonen. En applikasjon kan kjøre et metodekall og sende med en peker til seg selv. Denne klassen trenger dermed ikke tenke mer på hva som skjer før svaret kommer i form av et metodekall til den selv. Oppgavene kan dessuten utføres i forskjellige tråder. Dette er spesielt viktig i en klasse som styrer brukergrensesnittet. Hadde tråden måttet vente på svar ville man oppleve brukergrensesnittet som "fryst" inntil svaret kom. Dette løses ofte med å manuelt opprette flere tråder i andre programmeringsspråk.

4.1.5 Storyboards

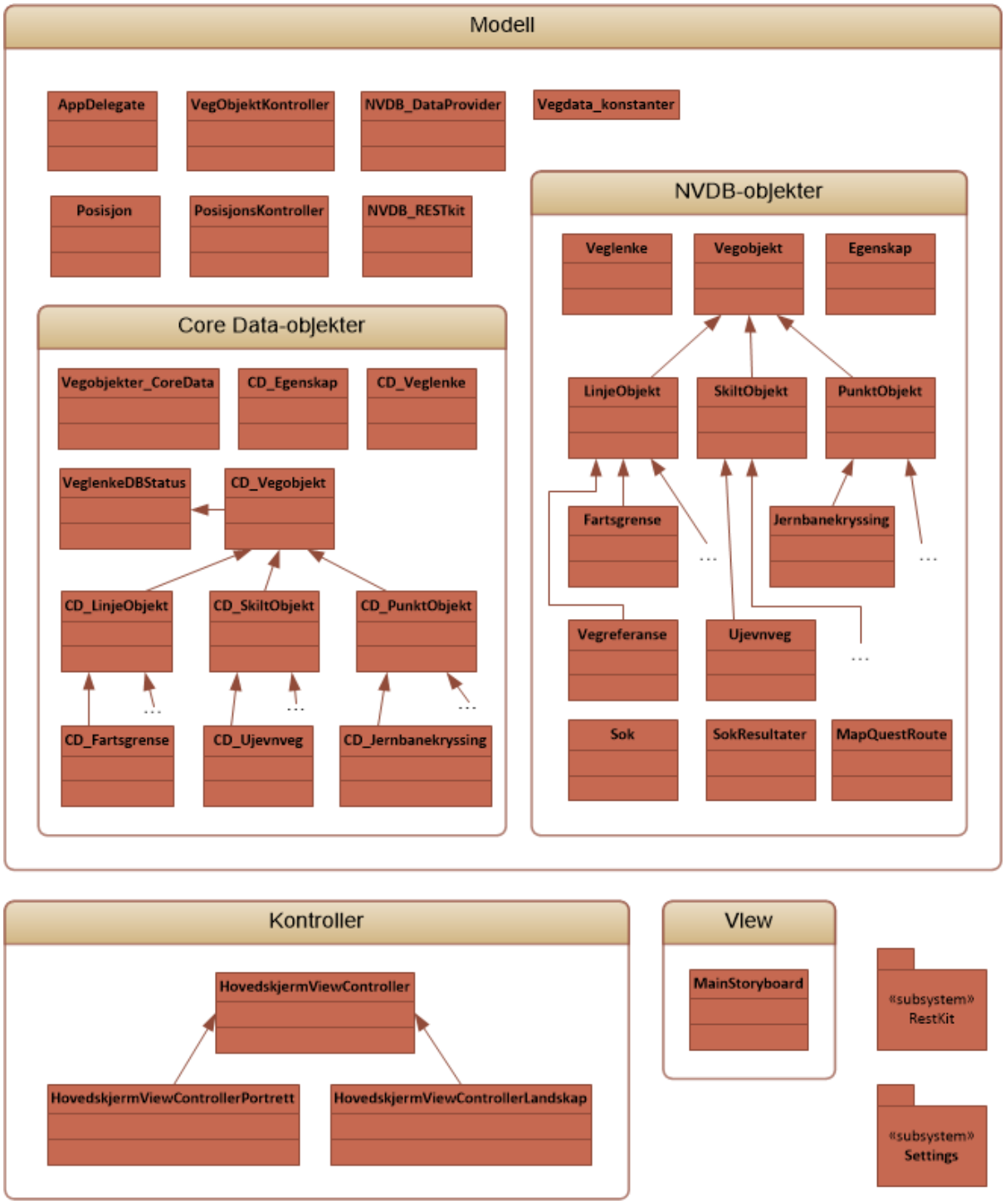
Et storyboard er en grafisk fremstilling av brukergrensesnittet i en iOS-applikasjon. Det viser skjermbildene i applikasjonen og koblingene mellom dem. I Xcode designes brukergrensesnittet i et eget grafisk storyboard-redigeringsprogram. Alt som gjøres i det grafiske redigeringsprogrammet konverteres automatisk til XML i en egen *.storyboard-fil.



Figur 10: Et eksempel på et storyboard i Xcode (Foto: Apple).

4.2 Klassene i Kjørehjelperen

Kjørehjelperen består av veldig mange klasser. Dette er først og fremst fordi alle objektene som vises på skjermen har en egen klasse, en egen Core Data-klasse og en egen container-klasse for å holde på et sett av objekter. I dette delkapittelet ser vi på klassenes viktigste funksjoner. Det forklares også kort hva hver enkelt metode gjør. Kapittelet bør derfor brukes som et oppslagsverk.



Figur 11: Figuren viser klassene i Kjørehjelperen.

4.2.1 AppDelegate

AppDelegate-klassen genereres automatisk av Xcode når man starter et nytt iOS-prosjekt. Klassen er som navnet tilsier en delegat, og mottar viktige hendelser i livssyklusen til applikasjonen. Den brukes gjerne til å utføre handlinger når applikasjonen starter, avsluttes eller minimeres.

Vi benytter AppDelegate til å lese inn innstillinger og klargjøre Core Data. Vi oppretter et NSManagedObjectContext-objekt[46] og sjekker størrelsen på SQLite-databasen. Hvis denne er større enn hva brukeren har satt som grense slettes de eldste innføringene inntil databasens størrelse er 90 % av denne. Når applikasjonen avsluttes kjøres det en ekstra lagring mot Core Data.

Første gang applikasjonen starter settes innstillingene i applikasjonen til standard-verdier. I tillegg vises det en dialogboks til brukeren med pålagte erklæringer fra NVDB og MapQuest.

Metodene i AppDelegate er:

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

Denne metoden setter standardinstillinger og viser pålagte erklæringer hvis applikasjonen kjøres for første gang.

- (BOOL)isForstegangsOppstart

Metoden sjekker om applikasjonen kjøres for første gang.

- (void)applicationWillTerminate:(UIApplication *)application

Kjører "saveContext"-metoden. Autogenerert.

- (void)saveContext

Lagrer endringer i Core Data. Autogenerert.

- (void)applicationDidBecomeActive:(UIApplication *)application

Sender en notifikasjon til andre klasser om at applikasjonen akkurat har blitt åpnet. I tillegg slettes data fra databasen hvis denne er for stor.

- (NSManagedObjectContext *)managedObjectContext

- (NSManagedObjectModel *)managedObjectModel

- (NSPersistentStoreCoordinator *)persistentStoreCoordinator

Disse tre metodene oppretter hver sine Core Data-relaterte objekter. Autogenererte.

- (NSNumber *)coreDataSize

Finner størrelsen på databasen.

- (NSURL *)applicationLibraryCachesDirectory

Returnerer stien til SQLite-filen.

[46] Les mer om hva NSManagedObjectContext-objektet brukes til under 3.1 Core Data.

4.2.2 Posisjon og PosisjonsKontroller

Posisjon er en klasse som samler den viktigste informasjonen fra telefonens posisjonstjeneste. Den inneholder egenskapene breddegrad, lengdegrad, hastighetIMeterISek, retning, meterOverHavet og presisjon. I tillegg inneholder klassen hjelpemetoden hastighetIKmT som konverterer hastigheten til km/t.

PosisjonsKontroller implementerer CLLocationManagerDelegate. Dette tvinger den til å implementere metodene locationManager:didUpdateLocations: og locationManager:didFailWithError:. PosisjonsKontroller mottar altså posisjoner fra enheten, oppretter et Posisjon-objekt og sender det videre med sin egen delegat, PosisjonDelegate.

PosisjonDelegate defineres altså i denne filen. Protokollen krever implementasjon av metodene posisjonOppdatering: og posisjonFeil:.

```
@protocol PosisjonDelegate
@required
- (void) posisjonOppdatering:(Posisjon *)posisjon;
- (void) posisjonFeil:(NSError *)feil;
@end

@interface PosisjonsKontroller : NSObject <CLLocationManagerDelegate>
@property (nonatomic, strong) CLLocationManager * lokMan;
@property (nonatomic, assign) id delegate;
@property (nonatomic, strong) CLLocation * sisteOppdatering;
@end

@interface Posisjon : NSObject
@property (nonatomic, strong) NSDecimalNumber * breddegrad;
@property (nonatomic, strong) NSDecimalNumber * lengdegrad;
@property (nonatomic, strong) NSDecimalNumber * hastighetIMeterISek;
@property (nonatomic, strong) NSDecimalNumber * retning;
@property (nonatomic, strong) NSDecimalNumber * meterOverHavet;
@property (nonatomic, strong) NSDecimalNumber * presisjon;
- (NSDecimalNumber *) hastighetIKmT;
@end
```

Kodesnutt 21: I Posisjon.h deklareres PosisjonDelegate og klassene PosisjonsKontroller og Posisjon.

Metodene i PosisjonsKontroller er:

`- (id) init`

Konstruktør, kalles når en instans av klassen opprettes. Initialiserer et CLLocationManager-objekt.

`- (void) locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations`

Mottar ny posisjon. Hvis enheten har beveget seg nok og det har gått nok tid siden forrige posisjon, basert på satte konstanter, opprettes det et Posisjon-objekt som sendes med delegatens posisjonOppdatering:-metode.

- (void) locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error

Kjøres hvis enheten ikke klarer å finne posisjonen. Kaller delegatens posisjonFeil:-metode.

- (void) oppdaterPresisjonMedFart:(NSDecimalNumber *)meterISekundet

Sjekker enhetens fart og oppdaterer deretter ønsket presisjon til å samsvare med ønsket tidsintervall, basert på satt konstant.

4.2.3 VegObjektKontroller

Oppgaven til VegObjektKontroller er å motta en posisjon, og deretter levere det som skal vises på denne posisjonen. Den deklarerer VegObjektDelegate som inneholder metodene vegObjekterErOppdatert: og avstandTilPunktobjekt:MedKey:. Disse brukes til å levere det som skal vises på skjermen basert på posisjonen. Den implementerer selv NVDBResponseDelegate som brukes til å motta svar på forespørsler fra NVDB_DataProvider.

Klassen sender en forespørsel videre for å finne en vegreferanse til posisjonen. Deretter oppretter den et søkeobjekt basert på denne vegreferansen som sendes videre. Den mottar så alle vegobjektene som er knyttet til denne vegen. Klassen går så gjennom disse objektene og plukker ut de som er aktuelle å vise. Samtidig sender den forespørsler videre for å finne avstanden til noen av disse. Til slutt sender den informasjon om hvilke objekter som skal vises med vegObjekterErOppdatert:-metoden og avstander fortløpende når de mottas med avstandTilPunktobjekt:MedKey:-metoden.

Metodene i VegObjektKontroller er:

- (id)initMedDelegate:(id)delegat OgManagedObjectContext:(NSManagedObjectContext *)context

Konstruktør. Setter delegaten som skal motta oppdateringene, oppretter en instans av NVDB_DataProvider og kaller settOppObjektreferanseArray-metoden.

+ (NSArray *)settOppObjektreferanseArray

Setter opp et array bestående av alle objekttypene som kan vises på skjermen i applikasjonen.

- (void)oppdaterMedBreddegrad:(NSDecimalNumber *)breddegrad OgLengdegrad:(NSDecimalNumber *)lengdegrad

Sender en forespørsel til NVDB_DataProvider etter en vegreferanse.

- (void)sokMedVegreferanse

Oppretter et søkeobjekt med vegreferanse og objekttyper. Sender deretter en forespørsel til

NVDB_DataProvider etter vegobjekter og sender med søkeobjektet og objektmappingen (se hentObjektMapping-metoden).

- (NSArray *)hentObjekttyper

Returnerer et array med Objekttype-objekter som brukes i søkeobjektet.

- (RKDynamicMapping *) hentObjektMapping

Returnerer et RKDynamicMapping-objekt som definerer hvordan RestKit skal mappe data til vegobjektene.

- (void) leggTilDataIDictionary:(NSMutableDictionary *)returDictionary FraSokeresultater:(SokResultater *)resultater MedAvstandsArray:(NSMutableArray *)avstand

Metoden sjekker om mottatt objekt skal vises til brukeren. Hvis brukeren befinner seg på linjen som defineres i et LinjeObjekt kalles leggTilLinjeDataIDictionary:MedVegObjekt:-metoden. Hvis objektet er et PunktObjekt eller et SkiltObjekt kjøres følgende test:

```
else if(self.forrigePosisjon &&
        self.forrigePosisjon.doubleValue >= 0 &&
        (naermestePosisjon.doubleValue < 0 ||
         [VegObjektKontroller diffMellomA:posisjon OgB:vLenke.fra].doubleValue <=
         [VegObjektKontroller diffMellomA:naermestePosisjon
                                    OgB:vLenke.fra].doubleValue) &&
        ((posisjon.doubleValue > self.forrigePosisjon.doubleValue &&
          vLenke.fra.doubleValue > posisjon.doubleValue &&
          ([obj isKindOfClass:[PunktObjekt class]] ||
           ([obj isKindOfClass:[SkiltObjekt class]] &&
            [((SkiltObjekt *)obj).ansiktsside
                              isEqualToString:SKILTPLATE_ANSIKTSSIDE_MED]))) ||
         (vLenke.fra.doubleValue < posisjon.doubleValue &&
          ([obj isKindOfClass:[PunktObjekt class]] ||
           ([obj isKindOfClass:[SkiltObjekt class]] &&
            [((SkiltObjekt *)obj).ansiktsside
                               isEqualToString:SKILTPLATE_ANSIKTSSIDE_MOT])))))
```

Kodesnutt 22: En komplisert test i leggTilDataIDictionary:FraSokeresultater:MedAvstandsArray:-metoden.

Dette er en veldig komplisert test. Forklaringen på testen er følgende:

A - [obj isKindOfClass:[PunktObjekt class]]

B - [obj isKindOfClass:[SkiltObjekt class]]

C - self.forrigePosisjon

D - self.forrigePosisjon.doubleValue >= 0

E - naermestePosisjon.doubleValue < 0

F - [VegObjektKontroller diffMellomA:posisjon OgB:vLenke.fra].doubleValue

<= [VegObjektKontroller diffMellomA:naermestePosisjon OgB:vLenke.fra].doubleValue

G - posisjon.doubleValue > self.forrigePosisjon.doubleValue

H - vLenke.fra.doubleValue > posisjon.doubleValue

I - vLenke.fra.doubleValue < posisjon.doubleValue

*J - [((SkiltObjekt *)obj).ansiktsside isEqualToString:SKILTPLATE_ANSIKTSSIDE_MED]*

*K - [((SkiltObjekt *)obj).ansiktsside isEqualToString:SKILTPLATE_ANSIKTSSIDE_MOT]*

Hvis vi kjører i stigende retning på en veglenke, og objektet enten er det første eller det nærmeste objektet til enhetens posisjon, samtidig som det har en høyere posisjon enn enheten (altså ligger foran enheten). Hvis objektet er et skiltobjekt må ansiktssiden være med metreringsretning[47]:

C && D && G && H && (E || F) && (A || (B && J))

Hvis vi kjører i synkende retning på en veglenke, og objektet enten er det første eller det nærmeste objektet til enhetens posisjon, samtidig som det har en lavere posisjon enn enheten (altså ligger foran enheten). Hvis objektet er et skiltobjekt må ansiktssiden være mot metreringsretning:

C && D && I && (E || F) && (A || (B && K))

Setter vi sammen disse får vi:

(C && D && G && H && (E || F) && (A || (B && J))) || (C && D && I && (E || F) && (A || (B && K)))

Med boolsk algebra finner vi at dette er ekvivalent med:

C && D && (E || F) && ((G && H && (A || (B && J))) || (I && (A || (B && K))))

Hvis testen er sann kalles enten leggTilPunktDataIDictionary:MedVegObjekt:OgAvstandsArray:-metoden eller leggTilSkiltDataIDictionary:MedVegObjekt:OgAvstandsArray:-metoden avhengig av om objektet er et PunktObjekt eller et SkiltObjekt.

- (void)leggTilLinjeDataIDictionary:(NSMutableDictionary *)returDictionary MedVegObjekt:(LinjeObjekt <VegobjektProtokoll> *)objekt

Metoden legger til navnet på LinjeObjekt-objektet i returDictionary-objektet[48]. Dette betyr at det skal vises for brukeren. Hvis det er nødvendig med mer informasjon, f.eks. hvilken fartsgrense det er, legges også dette til.

- (void)leggTilPunktDataIDictionary:(NSMutableDictionary *)returDictionary MedVegObjekt:(PunktObjekt <VegobjektProtokoll>

[47] Metreringsretningen er den stigende retningen på egn veglenke, altså når posisjonstallet går fra 0 mot 1.
[48] En dictionary er en datastruktur bestående av nøkkel- og verdipar.

***)objekt OgAvstandsArray:(NSMutableArray *)avstand**

- (void)leggTilSkiltDataIDictionary:(NSMutableDictionary *)returDictionary MedVegObjekt:(SkiltObjekt <VegobjektProtokoll> *)objekt OgAvstandsArray:(NSMutableArray *)avstand

Metodene legger til navnet på PunktObjekt- eller SkiltObjekt-objektet I returDictionary-objektet, og eventuell ekstra informasjon som er nødvendig. Metoden legger også til informasjon om objektets posisjon i avstand-objektet.

- (NSDecimalNumber *)kalkulerVeglenkePosisjon

Metoden finner enhetens posisjon på en veglenke, representert ved et nummer mellom 0 og 1. Les mer om posisjonering på veglenker under "1.2.2.1 Finne ut hvor brukeren befinner seg".

+ (NSDecimalNumber *)diffMellomA:(NSDecimalNumber *)desimalA OgB:(NSDecimalNumber *)desimalB

Metoden regner ut differansen mellom to desimaltall.

- (void)svarFraNVDBMedResultat:(NSArray *)resultat OgVeglenkeId:(NSNumber *)lenkeId

Metode definert i NVDBResponseDelegate. Metoden kalles når et svar fra NVDB_DataProvider er klart. Hvis svaret inneholder en vegreferanse kalles sokMedVegreferanse-metoden. Hvis svaret inneholder vegobjekter kjøres leggTilDataIDictionary:FraSokeresultater:MedAvstandsArray:-metoden for hvert objekt. Forespørsler om avstander sendes herfra for de objektene det gjelder. Til slutt kalles delegatens vegObjekterErOppdatert-metode med returDictionary-objektet.

- (void)svarFraMapQuestMedResultat:(NSArray *)resultat OgKey:(NSString *)key

Metode definert i NVDBResponseDelegate. Metoden kalles når et svar fra NVDB_DataProvider er klart. Delegatens avstandTilPunktobjekt:MedKey:-metode kalles med resultatet.

- (NSMutableDictionary *) opprettReturDictionaryMedDefaultVerdier

Metoden oppretter et NSMutableDictionary-objekt og setter standardverdier for alle vegobjektene.

4.2.4 NVDB_DataProvider

Oppgaven til NVDB_DataProvider er å levere dataene den blir bedt om. Den fungerer som et abstrakt lag som leverer data fra NVDB og MapQuest. Klasser som ber denne klassen om data trenger ikke tenke på hvor dataene kommer fra. NVDB_DataProvider leverer data fra Core Data hvis de eksisterer, ellers henter den data fra NVDB. Den sørger også for å lagre nye data til Core Data så de kan hentes derfra neste gang.

```
@interface NVDB_DataProvider : NSObject <NVDBResponseDelegate>
@property (nonatomic, strong) NSManagedObjectContext * managedObjectContext;

- (id)initMedManagedObjectContext:(NSManagedObjectContext *)context
                       OgAvsender:(NSObject *)aAvsender;
- (void)hentVegreferanseMedBreddegrad:(NSDecimalNumber *)breddegrad
                          OgLengdegrad:(NSDecimalNumber *)lengdegrad;
- (void)hentVegObjekterMedSokeObjekt:(Sok *)sok OgMapping:(RKMapping *)mapping;
- (void)hentAvstandmedKoordinaterAX:(NSDecimalNumber *)ax
                                 AY:(NSDecimalNumber *)ay
                                 BX:(NSDecimalNumber *)bx
                                 BY:(NSDecimalNumber *)by
                              ogKey:(NSString *)key;
@end
```

Kodesnutt 23: Headerfilen til NVDB_DataProvider. Selv om implementasjonsfilen er på over 1200 linjer og inneholder mange metoder, er det kun fire metoder som deklareres i headerfilen og dermed er synlige utenfor klassen.

Metodene i NVDB_DataProvider er :

- (id)initMedManagedObjectContext:(NSManagedObjectContext *)context OgAvsender:(NSObject *)aAvsender

Konstruktør. Mottar Core Data-konteksten og oppretter en instans av NVDB_RESTkit.

- (void)hentVegreferanseMedBreddegrad:(NSDecimalNumber *)breddegrad OgLengdegrad:(NSDecimalNumber *)lengdegrad

Spør videre mot NVDB_RESTkit etter å ha satt opp parametere og mapping for søk etter vegreferanse.

- (void)hentVegObjekterMedSokeObjekt:(Sok *)sok OgMapping:(RKMapping *)mapping

Sjekker om det finnes data for vegen i Core Data. Hvis det gjør det kalles hentVegObjekterFraCoreDataMedVeglenkeCDObjekt:-metoden. Hvis ikke kalles hentVegObjekterFraNVDBMedSokeObjekt:OgMapping:-metoden.

- (void)hentAvstandmedKoordinaterAX:(NSDecimalNumber *)ax AY:(NSDecimalNumber *)ay BX:(NSDecimalNumber *)bx BY:(NSDecimalNumber *)by ogKey:(NSString *)key

Spør videre mot NVDB_RESTkit etter å ha satt opp parametere og mapping for søk etter avstand med MapQuest.

- (void)hentVegObjekterFraNVDBMedSokeObjekt:(Sok *)sok OgMapping:(RKMapping *)mapping

Spør videre mot NVDB_RESTkit etter å ha satt opp parametere og mapping for søk etter vegobjekter.

- (void)hentVegObjekterFraCoreDataMedVeglenkeCDObjekt:
(VeglenkeDBStatus *)vlenke

Henter ut alle vegobjektene i Core Data for gitt veglenke. Objektene gjøres om fra Core Data-objekter til NVDB-objekter og settes inn i samme datastruktur som returneres fra NVDB_RESTkit-klassen. Kaller deretter delegatens svarFraNVDBMedResultat:OgVeglenkeId:-metode.

- (void)svarFraNVDBMedResultat:(NSArray *)resultat
OgVeglenkeId:(NSNumber *)lenkeId

Hvis mottatt svar inneholder vegobjekter lagres disse til Core Data. Dette gjøres ved å konvertere NVDB-objektene til Core Data-objekter. Skiltplater gjøres også om til NVDB- og Core Data-objekter. Til slutt kalles delegatens svarFraNVDBMedResultat:OgVeglenkeId:-metode.

- (void)svarFraMapQuestMedResultat:(NSArray *)resultat
OgKey:(NSString *)key

Sender svaret direkte videre ved å kalle delegatens svarFraMapQuestMedResultat:OgKey:-metode.

+ (NSArray *)gjorOmTilSkiltobjekterMedResultat:(NSArray *)resultat

Gjør om mottatte skiltplater til NVDB-objekter.

+ (NSDictionary
***)parametereForKoordinaterMedBreddegrad:(NSDecimalNumber**
***)breddegrad OgLengdegrad:(NSDecimalNumber *)lengdegrad**

Konverterer koordinater til URL-parametere i en NSDictionary.

+ (NSDictionary
***)parametereForBoundingBoxMedBreddegrad:(NSDecimalNumber**
***)breddegrad OgLengdegrad:(NSDecimalNumber *)lengdegrad**

Konverterer koordinater til parametere oppgitt som en bounding box i en NSDictionary. Metoden benyttes ikke i siste versjon av Kjørehjelperen.

+ (NSDictionary *)parametereForMapQuestAvstandMedAX:(NSDecimalNumber
***)ax AY:(NSDecimalNumber *)ay BX:(NSDecimalNumber *)bx**
OgBY:(NSDecimalNumber *)by

Konverterer koordinater til parametere i en NSDictionary.

+ (NSDictionary *)parametereForSok:(Sok *)sok

Konverterer et søkeobjekt til parametere.

4.2.5 NVDB_RESTkit

NVDB_RESTkit er ansvarlig for kommunikasjon med web-APIene. Den benytter RestKit til å gjøre spørringer mot det aktuelle APIet.

```
@protocol NVDBResponseDelegate
@required
- (void) svarFraNVDBMedResultat:(NSArray *)resultat
                    OgVeglenkeId:(NSNumber *)lenkeId;
- (void) svarFraMapQuestMedResultat:(NSArray *)resultat OgKey:(NSString *)key;
@end

@interface NVDB_RESTkit : NSObject

@property (nonatomic, assign) id delegate;

- (void) hentDataMedURI:(NSString *)uri
             Parametere:(NSDictionary *)parameter
                Mapping:(RKMapping *)mapping
                KeyPath:(NSString *)keyPath
           OgVeglenkeId:(NSNumber *)lenkeId;
- (void) hentAvstandMellomKoordinaterMedParametere:(NSDictionary *)parameter
                                           Mapping:(RKMapping *)mapping
                                             OgKey:(NSString *)key;
@end
```

Kodesnutt 24: I NVDB_RESTkit.h deklareres en protokoll som brukes til å sende svar på mottatte forespørsler når de er klare. NVDB_RESTkit inneholder to metoder, en som spør mot NVDB og en som spør mot MapQuest.

Begge metodene deklarert i headerfilen (se kodesnutt 24) bruker RestKit til å spørre mot APIer. De initialiserer RestKit, setter opp nødvendig info og mapping og utfører deretter spørringen. Når de mottar svar kjører de delegatens svarFra…-metode. Den siste parameteren i begge metodene, "lenkeId" og "key" brukes ikke i metoden, men sendes tilbake med svaret for at svaret skal kunne identifiseres med riktig spørring.

4.2.6 Vegdata_konstanter

Vegdata_konstanter er ikke en klasse, men en fil som definerer konstanter som brukes i Kjørehjelperen. Det benyttes mye testing på strenger, både i Kjørehjelperen og i NVDB APIet. Det er derfor viktig å ha disse definert på ett sted for å unngå feil og å gjøre endringer vesentlig lettere.

```
#define INGEN_OBJEKTER @"-1"
#define INGEN_EGENSKAPER @"Ingen egenskaper"
#define SKILLETEGN_SKILTOBJEKTER @"#"

#define VEGREFERANSE_BREDDEGRAD @"breddegrad"
#define VEGREFERANSE_LENGDEGRAD @"lengdegrad"

#define FARTSGRENSE_ID 105
#define FARTSGRENSE_KEY @"fartsgrense"
#define FARTSGRENSE_CD @"CD_Fartsgrense"
#define FARTSGRENSE_FART_KEY @"Fartsgrense"
#define FARTSGRENSE_BRUKERPREF @"skilt_fartsgrense"
```

Kodesnutt 25: Et utdrag fra Vegdata_konstanter. Konstantene brukes både til sammenlikninger i Kjørehjelperen, tilknytning av objekter til Core Data og brukerpreferanser, og til spørringer mot NVDB APIet.

4.2.7 NVDB-objekter

NVDB-objektene er objekter som brukes til å mappe data til eller fra JSON med RestKit. Sok-klassen definerer objektet som sendes med spørringer til NVDB APIet. MapQuestRoute er objektet svar fra MapQuest mappes til. De andre objektene er stort sett vegobjekter som data fra NVDB mappes til.

Vegobjektene er subklasser av enten LinjeObjekt, PunktObjekt eller SkiltObjekt. Disse klassene er igjen subklasser av Vegobjekt hvor de viktigste egenskapene defineres. Vegobjekt inneholder blant annet to arrayer med Egenskap- og Veglenke-objekter. Grunnen til at vegobjektene arver fra enten LinjeObjekt, PunktObjekt eller SkiltObjekt er at disse typene behandles ulikt i VegObjektKontroller-klassen. Man kan dermed teste hvilken av disse typene objektet er en subklasse av, og dermed slippe å teste på alle mulige NVDB-objekttyper.

```
@protocol VegobjektProtokoll <NSObject>
@required
+ (RKObjectMapping *)mapping;
+ (NSArray *)filtere;
+ (NSNumber *)idNr;
+ (NSString *)key;
+ (BOOL)objektSkalVises;
@end

@interface Vegobjekt : NSObject
@property (nonatomic, strong) NSArray * egenskaper;
@property (nonatomic, strong) NSArray * veglenker;
@property (nonatomic, strong) NSString * lokasjon;
+ (RKObjectMapping *)standardMappingMedKontainerKlasse:(Class)kontainerklasse;
@end

@interface LinjeObjekt : Vegobjekt
@property (nonatomic, strong) NSNumber * strekningsLengde;
@end

@interface PunktObjekt : Vegobjekt
@end
```

Kodesnutt 26: I Vegobjekt.h defineres VeobjektProtokoll, Vegobjekt, LinjeObjekt og PunktObjekt. SkiltObjekt defineres i en annen fil (SkiltObjekt.h).

VegobjektProtokoll implementeres av de fleste vegobjektene. Denne protokollen sørger for at klassene implementerer metoder som returnerer info om objektet til RestKit. Man kan dermed behandle alle vegobjekter likt uten å vite hva slags objekt det er snakk om.

```
@implementation Fartsgrense

@synthesize strekningsLengde, egenskaper, veglenker, lokasjon;

- (NSString *)hentFartFraEgenskaper
{
    for (Egenskap * e in egenskaper)
    {
        if([e.navn isEqualToString:FARTSGRENSE_FART_KEY])
        {
            return e.verdi;
```

```
        }
    }
    return INGEN_OBJEKTER;
}

+ (RKObjectMapping *)mapping
{
    return [self standardMappingMedKontainerKlasse:[Fartsgrenser class]];
}

+ (NSArray *)filtere {return nil;}

+ (NSNumber *)idNr {return [[NSNumber alloc] initWithInt:FARTSGRENSE_ID];}

+ (NSString *)key {return FARTSGRENSE_KEY;}

+ (BOOL)objektSkalVises
{
    return [[NSUserDefaults standardUserDefaults]
             boolForKey:FARTSGRENSE_BRUKERPREF];
}

@end
```

Kodesnutt 27: Fartsgrense arver fra LinjeObjekt og implementerer VegobjektProtokoll. Vi trenger derfor ikke deklarere egenskapene strekningsLengde, egenskaper, veglenker og lokasjon i headerfilen. Vi må implementere metodene fra VegobjektProtokoll som vi ser at er gjort her i implementasjonsfilen.

Metodene i et standard vegobjekt er:

`+ (RKObjectMapping *)mapping`

Returnerer mappingen som brukes for å mappe JSON-data til objektet med RestKit.

`+ (NSArray *)filtere`

Hvis det benyttes filter[49] ved søk etter objektet i NVDB returneres dette av denne metoden.

`+ (NSNumber *)idNr`

Returnerer identifikasjonsnummeret til objektet i NVDB. Nummeret er definert i Vegdata_konstanter.

`+ (NSString *)key`

Returnerer strengen som brukes til å identifisere objektet innad i Kjørehjelperen.

`+ (BOOL)objektSkalVises`

Returnerer hvorvidt objektet skal vises for brukeren eller ikke, basert på brukerinnstillingene i applikasjonen.

4.2.8 SokResultater

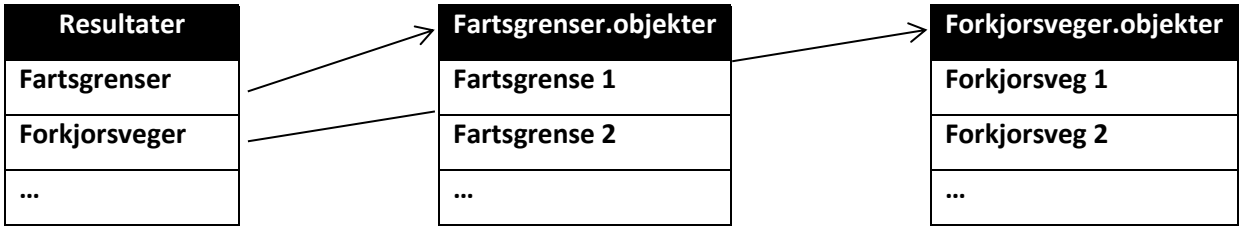
```
@interface SokResultater : NSObject
@property (nonatomic, strong) NSArray * objekter;
-(id)initMedObjekter:(NSArray *)aObjekter;
```

[49] Les om bruk av filter i NVDB under "1.2.2.2 Finne objektene på den vegen vi befinner oss".

```
@end

@interface Fartsgrenser : SokResultater
@end

@interface Forkjorsveger : SokResultater
@end

...
```

Kodesnutt 28: SokResultater.h. Her ser vi SokResultater og to av subklassene, Fartsgrenser og Forkjorsveger.

SokResultater er en klasse som kun inneholder et array. Den har en subklasse for hver enkelt NVDB-objekttype. Klassen Fartsgrenser inneholder for eksempel kun et array bestående av Fartsgrense-objekter. Grunnen til at vegobjektene pakkes inn i sine respektive kontainerklasser i stedet for et vanlig array er at dette muliggjør testing på type på et høyere nivå. Det lar oss også mappe mange objekter til en sortert struktur i RestKit.



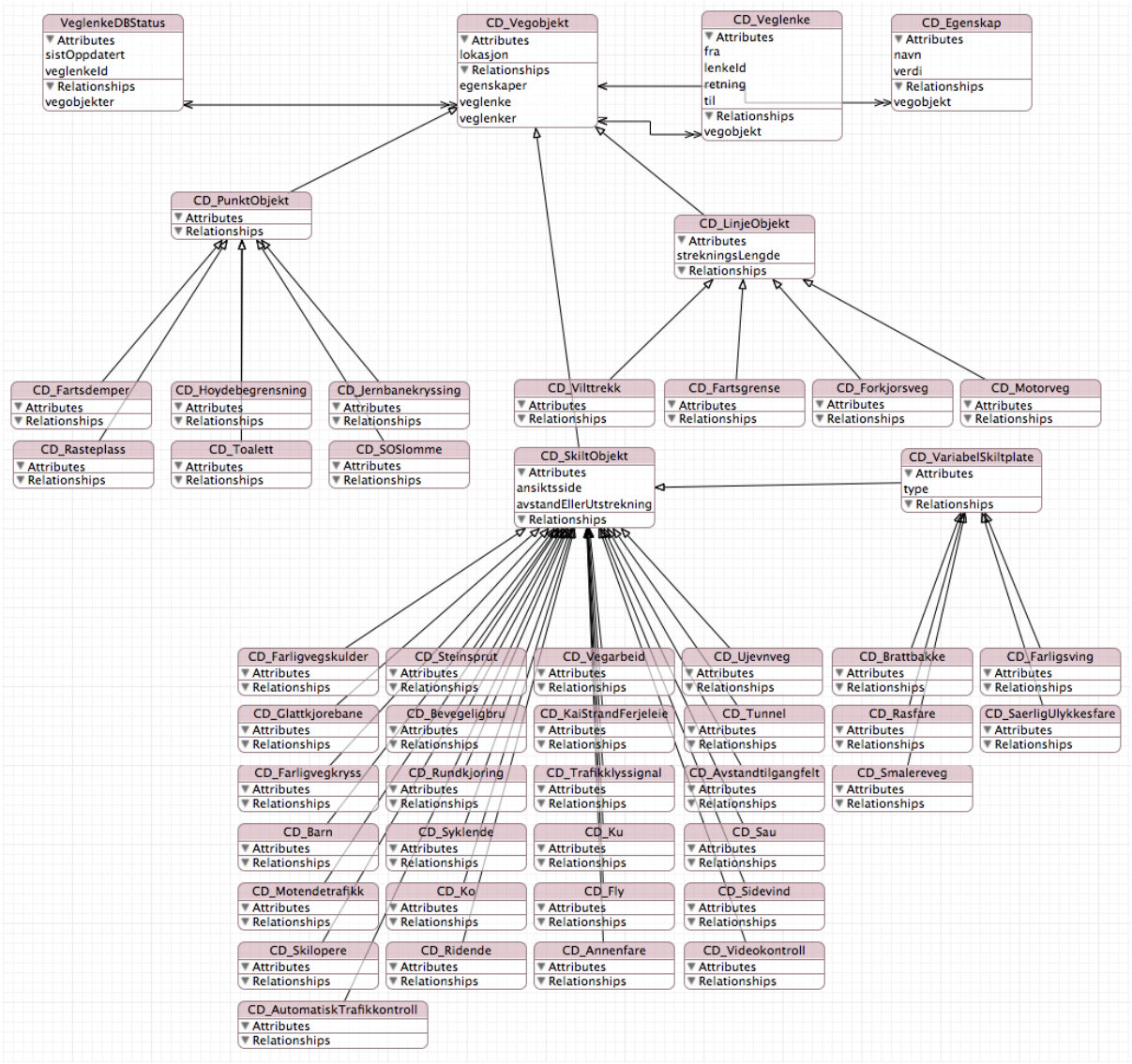
Figur 12: Arrayet "Resultater" som returneres av RestKit inneholder et SokResultater-objekt for hver av NVDB-objektene. Disse objektene inneholder igjen et array med sine respektive Vegobjekt-objekter.

4.2.9 Core Data-objekter

Core Data-objektene er autogenerert av Core Data. De speiler entitetene og forholdet mellom dem. Entitetene speiler igjen NVDB-objektene. Unntaket er VeglenkeDBStatus som representerer én veglenke, et tidspunkt for oppdatering og et sett med alle de tilhørende vegobjektene.

```
@interface VeglenkeDBStatus : NSManagedObject
@property (nonatomic, retain) NSDate * sistOppdatert;
@property (nonatomic, retain) NSNumber * veglenkeId;
@property (nonatomic, retain) NSSet * vegobjekter;
@end

@interface VeglenkeDBStatus (CoreDataGeneratedAccessors)
- (void)addVegobjekterObject:(CD_Vegobjekt *)value;
- (void)removeVegobjekterObject:(CD_Vegobjekt *)value;
- (void)addVegobjekter:(NSSet *)values;
- (void)removeVegobjekter:(NSSet *)values;
@end
```

Kodesnutt 29: Den autogenererte headerfilen til VeglenkeDBStatus.



Figur 13: Den grafiske fremstillingen av entitetene og relasjonene mellom dem i Core Data-datamodellen i Kjørehjelperen.

I Kjørehjelperen er NVDB- og Core Data-objektene separate. Det konverteres manuelt fra den ene typen til den andre ved lasting og lagring. Tidsbegrensninger førte til denne implementasjonen i Kjørehjelperen, men man kunne implementert vegobjekter som både lot seg mappe med RestKit og lagre til Core Data.

```
@interface CD_Vegobjekt : NSManagedObject
@property (nonatomic, retain) NSString * lokasjon;
@property (nonatomic, retain) NSSet *egenskaper;
@property (nonatomic, retain) VeglenkeDBStatus *veglenke;
@property (nonatomic, retain) NSSet *veglenker;
@end

@interface CD_Vegobjekt (CoreDataGeneratedAccessors)
- (void)addEgenskaperObject:(CD_Egenskap *)value;
- (void)removeEgenskaperObject:(CD_Egenskap *)value;
- (void)addEgenskaper:(NSSet *)values;
- (void)removeEgenskaper:(NSSet *)values;
```

```
- (void)addVeglenkerObject:(CD_Veglenke *)value;
- (void)removeVeglenkerObject:(CD_Veglenke *)value;
- (void)addVeglenker:(NSSet *)values;
- (void)removeVeglenker:(NSSet *)values;
@end
```

Kodesnutt 30: Headerfilen til CD_Vegobjekt.

4.2.10 HovedskjermViewController

HovedskjermViewController styrer hele applikasjonen. Den oppretter instanser av PosisjonsKontroller og VegObjektKontroller. Videre har den kontroll over alle de grafiske elementene på skjermen. Klassen mottar posisjoner fra PosisjonsKontroller og sender disse videre til VegObjektKontroller. Den mottar deretter informasjon om vegobjekter og viser dette på skjermen.

HovedskjermViewControllerPortrett og HovedskjermViewControllerLandskap er to subklasser av HovedskjermViewController. De inneholder altså de samme metodene og egenskapene som sin forelder. Kjørehjelperen startes i "Hoved…Portrett". Samtidig som denne gjør forelderens oppgaver, lytter den samtidig etter orienteringsendringer. Første gang den oppdager at telefonen er i landskapsmodus oppretter den "Hoved…Landskap". Deretter flyttes data og pekere frem og tilbake mellom disse to avhengig av telefonens orientering.

Grunnen til at Kjørehjelperen har en egen ViewController for hver orientering er at viewene, altså skjermbildene, er ganske forskjellige. Apple anbefaler[50] i slike tilfeller å bruke to ViewControllere i stedet for å flytte på elementer programmatisk.

HovedskjermViewController inneholder følgende metoder:

- (void)viewDidLoad

Kjøres når applikasjonen starter. Initialiserer klasser og strukturer, blant annet opprettes VegObjektKontroller og PosisjonsKontroller.

-(BOOL)shouldAutorotate

Forteller applikasjonen om den skal skifte orientering automatisk. Hvis HUD[51] er aktivert svarer metoden NO.

- (void)settOppLayoutArray

Setter opp et flerdimensjonalt array som holder på alle GUI-elementene[52] i applikasjonen.

[50] http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/RespondingtoDeviceOrientationChanges/RespondingtoDeviceOrientationChanges.html

[51] Head-up display eller heads-up display, også kjent som HUD, er en gjennomsiktig skjerm som viser informasjon uten at brukerne må se bort fra sine vanlige synspunkter. Opprinnelsen til navnet stammer fra at en pilot skal kunne se informasjon med hodet "opp" og se frem, i stedet for å se ned på instrumentene i cockpiten.

[52] Graphical user interface.

- (IBAction)hudKnappTrykket:(UISwitch *)knapp

Kjøres når HUD skrus av eller på. Speilvender alle elementer eller setter dem tilbake til normalt perspektiv.

- (BOOL)erFireTommerRetina

Forteller om telefonen har en 4" skjerm eller ikke.

- (BOOL)skalViseHUDKnapp

Leser fra brukerinnstillingene og forteller om HUD-knappen skal vises på skjermen eller ikke.

- (void) posisjonOppdatering:(Posisjon *)posisjon

Mottar en ny posisjon som sendes videre til VegObjektKontroller.

- (void) posisjonFeil:(NSError *)feil

Mottar feilmelding fra posisjonstjenesten og viser en feilmelding til brukeren.

- (void) vegObjekterErOppdatert:(NSDictionary *)data

Mottar en NSDictionary med data som skal vises til brukeren. Fjerner gamle data og fyller alle plassene på skjermen med ny data og grafikk.

- (void)avstandTilPunktobjekt:(NSDecimalNumber *)avstand MedKey:(NSString *)key

Mottar en avstand til et Punkt- eller SkiltObjekt. Finner riktig objekt på skjermen og viser avstanden.

+ (NSString *)finnTypeMedStreng:(NSString *)streng

Hjelpemetode som tolker en streng tilhørende et SkiltObjekt sendt fra VegObjektKontroller.

+ (NSString *)finnAvstandstekstMedStreng:(NSString *)streng ErVariabeltSkilt:(BOOL)erVariabelt

Hjelpemetode som tolker en streng tilhørende et SkiltObjekt sendt fra VegObjektKontroller.

I tillegg inneholder HovedskjermViewControllerPortrett følgende metoder:

- (void)viewDidLoad

Kaller forelderens metode med samme navn, og setter bakgrunnsbilde på skjermen.

- (void)awakeFromNib

Som viewDidLoad kalles også denne metoden automatisk når applikasjonen starter. I denne metoden settes det opp et NSNotificationCenter som lytter etter endringer i orienteringen.

- (void)orienteringEndret:(NSNotification *)notifikasjon

Kalles når orienteringen endres. Oppretter HovedskjermViewControllerLandskap ved første kjøring, overfører deretter data og pekere til den ViewControlleren som nå skal vises for brukeren.

-(NSUInteger)supportedInterfaceOrientations

Returnerer orienteringene som støttes av denne ViewControlleren, altså portrett.

- (void)hudKnappTrykket:(UISwitch *)knapp

Kalles når HUD-knappen trykkes. Siden HUD-modus alltid vises i landskapsmodus endrer denne metoden til landskapsmodus, og kaller HovedskjermViewControllerLandskaps hudKnappTrykket:-metode.

HovedskjermViewControllerLandskap inneholder i tillegg følgende metoder:

- (void)viewDidLoad

Kaller settOppLayoutArray:-metoden og setter bakgrunnsbilde på skjermen.

- (void)hudKnappTrykket:(UISwitch *)knapp

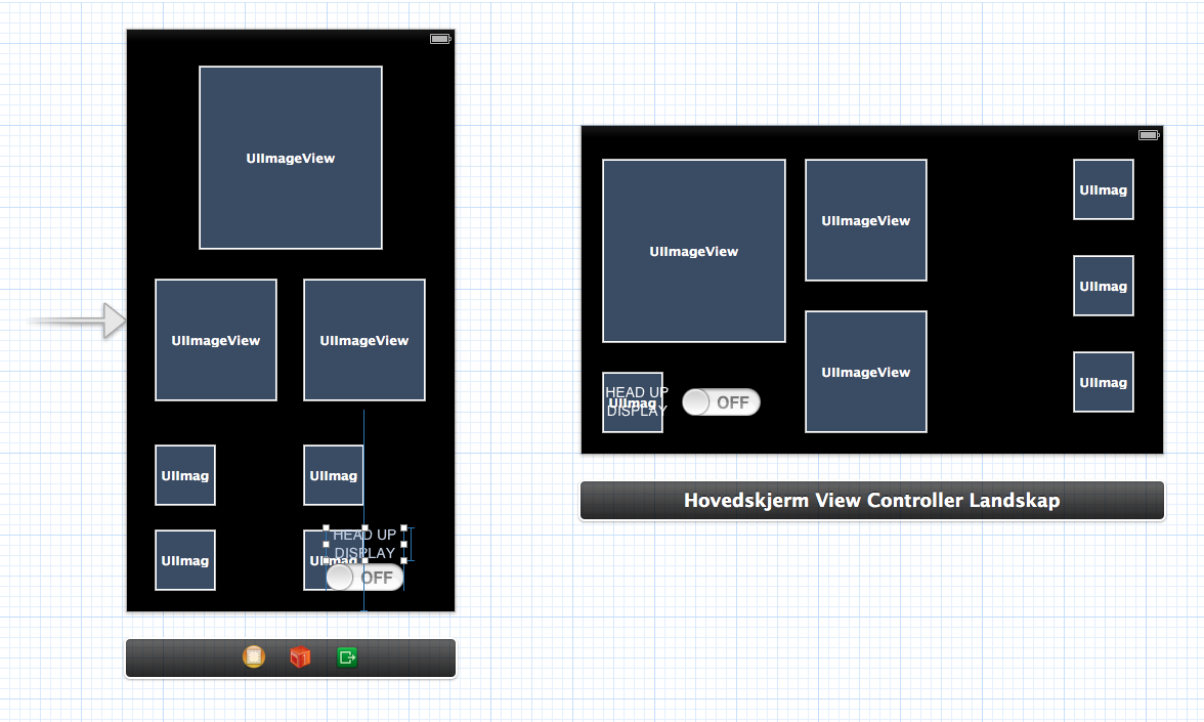
Kaller forelderens hudKnappTrykket:-metode og endrer bakgrunnsbildet.

-(NSUInteger)supportedInterfaceOrientations

Returnerer orienteringene som støttes av denne ViewControlleren, altså landskap.

4.2.11 MainStoryboard

Storyboardet inneholder informasjon om viewene i applikasjonen. Det består som sagt av autogenerert XML basert på en grafisk editor i Xcode.



Figur 14: Figuren viser den grafiske storyboard-editoren i Xcode og de to viewene for portrett- og landskapsmodus.

4.2.12 Settings

Under Settings.bundle/ ligger Root.plist. Her kan man endre hvilke innstillinger som gjelder for applikasjonen. I likhet med MainStoryboard består denne av autogenerert XML basert på en editor i Xcode. Denne editoren er tekstbasert, men tilbyr et forenklet grensesnitt for brukeren, i forhold til å redigere innholdet. Her er det feltet Key som benyttes når en innstilling så skal leses og benyttes av selve applikasjonen. Det finnes flere typer innstillinger, men vi benytter kun to: ToggleSwitch (bryter) og TextField (tekstfelt). I tillegg benytter vi Group (gruppe), men dette fungerer kun som en grupperingsmekanisme for innstillingene, og gir en overskrift for gruppen.

Key	Type	Value
iPhone Settings Schema	Dictionary	(3 items)
Settings Page Title	String	RootPList
Preference Items	Array	(46 items)
Item 0 (Group – Generelt)	Dictionary	(2 items)
Item 1 (Toggle Switch –	Dictionary	(4 items)
Type	String	Toggle Switch
Title	String	Lydvarsling
Identifier	String	lydvarsling
Default Value	Boolean	YES
Item 2 (Toggle Switch – HUD–	Dictionary	(4 items)
Item 3 (Text Field –	Dictionary	(5 items)
Item 4 (Group – Skilt (generelt))	Dictionary	(2 items)
Item 5 (Toggle Switch –	Dictionary	(4 items)
Item 6 (Toggle Switch –	Dictionary	(4 items)
Item 7 (Toggle Switch –	Dictionary	(4 items)
Item 8 (Toggle Switch –	Dictionary	(4 items)
Item 9 (Toggle Switch –	Dictionary	(4 items)
Item 10 (Toggle Switch –	Dictionary	(4 items)

Figur 15: Grensesnittet for opprettelse av brukerinnstillinger i Xcode.

Når en innstilling skal leses og benyttes i applikasjonen hentes den ut på denne måten:

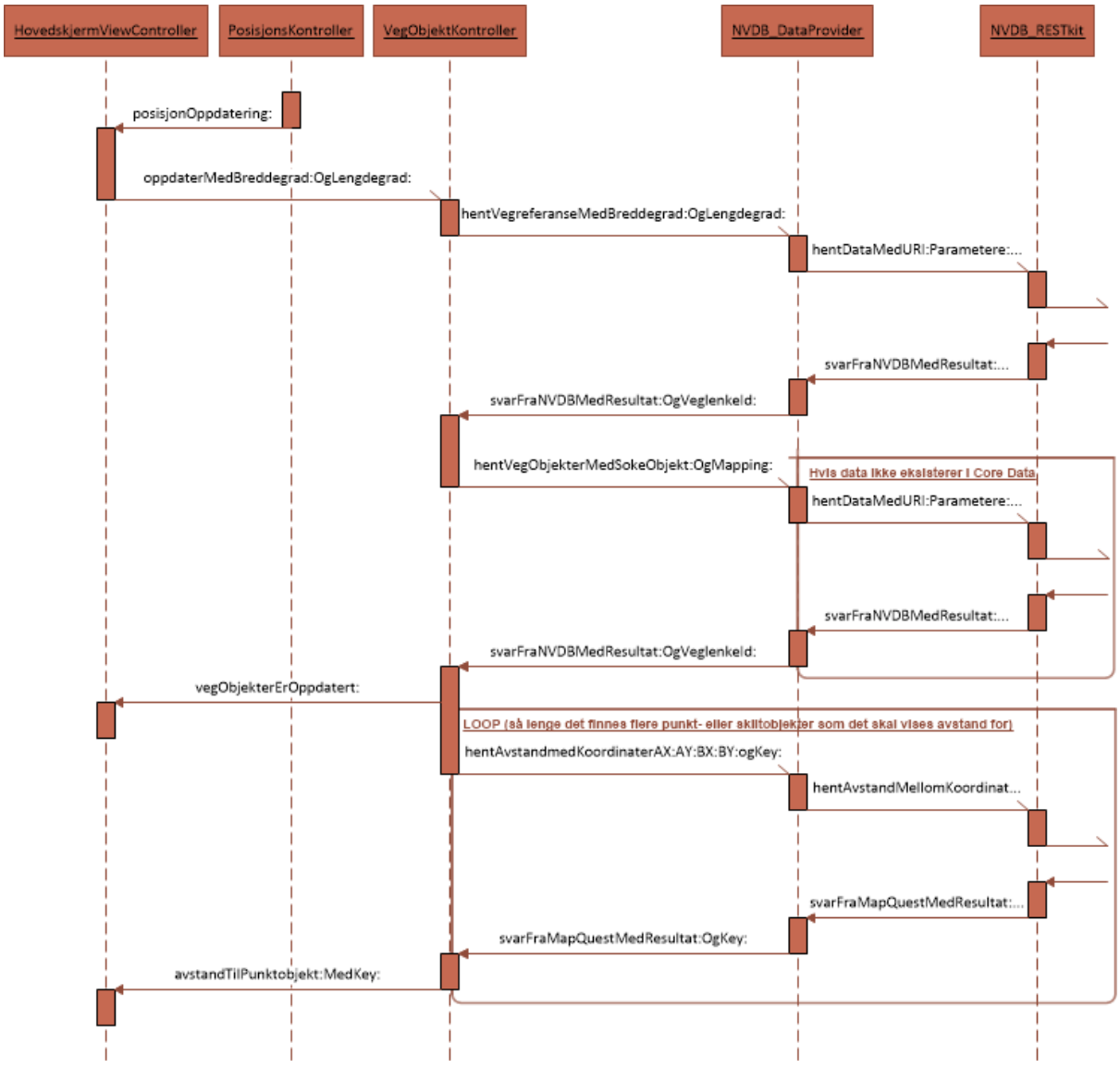
*NSString * enInnstilling = [[NSUserDefaults standardUserDefaults] stringForKey:@"keyTilInnstilling"];*

Innstillinger kan også leses ut som bool, int, osv. Da bytter man enkelt ut stringForKey med boolForKey, osv.

4.3 Gangen i applikasjonen

Når applikasjonen starter kjøres *application:didFinishLaunchingWithOptions:* og *applicationDidBecomeActive:* i AppDelegate-klassen. Deretter kjøres *viewDidLoad* i HovedskjermViewController-klassen. Les mer om hva som gjøres i disse metodene i "4.2.1 AppDelegate" og "4.2.10 HovedskjermViewController".

Nå er Kjørehjelperen klar til å levere data til brukeren. Det første som skjer er at PosisjonsKontroller leverer et nytt Posisjon-objekt til HovedskjermViewController. Denne klassen sender deretter Posisjon-objektet videre til VegObjektKontroller og er ferdig med sin oppgave inntil videre.



Figur 16: Sekvensdiagrammet viser en normalkjøring fra HovedskjermViewController mottar en posisjon til alle data vises på skjermen for brukeren.

VegObjektKontroller sender posisjonen videre til NVDB_DataProvider for å få en vegreferanse knyttet til posisjonen. NVDB_DataProvider sender dette videre til NVDB_RESTkit, som igjen sender en forespørsel til NVDB APIet. Siden det benyttes delegater er det ingen klasser som venter på svar. De våkner først når delegatens metode kalles.

Når RESTkit mottar et svar fra NVDB APIet mappes dette automatisk til et Vegreferanse-objekt, og sendes til NVDB_DataProvider. NVDB_DataProvider sender svaret videre til VegObjektKontroller. VegObjektKontroller bygger deretter et søkeobjekt med den aktuelle veglenke-id'en og alle vegobjektene med deres respektive mapping. Søkeobjektet sendes så til NVDB_DataProvider.

Når NVDB_DataProvider mottar søkeobjektet sjekker klassen om det eksisterer data for denne veglenken i Core Data og om den eventuelt er utdatert (se figur 16). Hvis data eksisterer og ikke er utdatert lastes dette ut av Core Data og sendes tilbake til VegObjektKontroller. Ellers sendes forespørselen videre til NVDB_RESTkit.

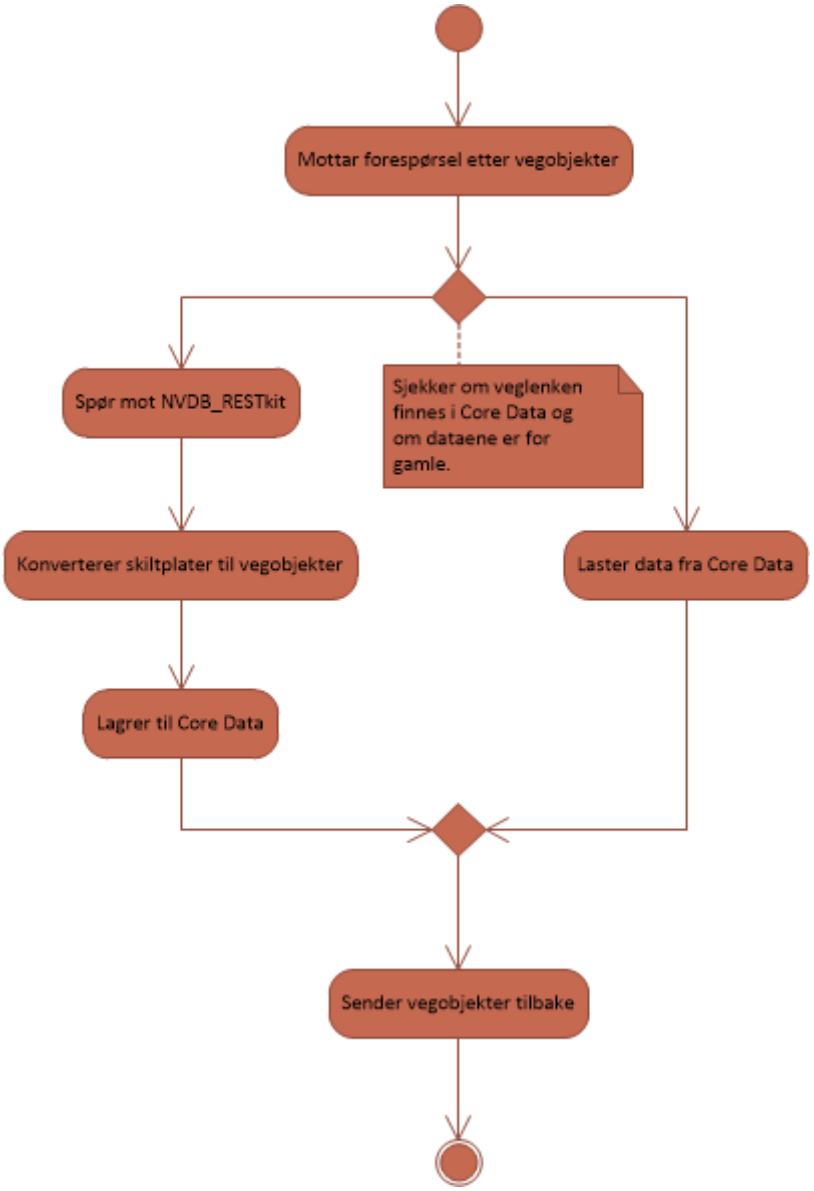
NVDB_RESTkit sender forespørselen videre til NVDB APIet. Når svar mottas, mappes det til vegobjekter og sendes tilbake til NVDB_DataProvider. Når NVDB_DataProvider mottar dette gjøres først skiltobjektene om til vegobjekter, før alt lagres til Core Data. Så sendes resultatet tilbake til VegObjektKontroller.

Når VegObjektKontroller mottar vegobjektene vet ikke klassen noe om de kom fra Core Data eller NVDB APIet. Den løper gjennom dataene og finner frem til det som skal vises til brukeren. Dette legges i en NSDictionary og returneres til HovedskjermViewController. Samtidig kjøres det en forespørsel mot NVDB_DataProvider for å finne avstanden til alle Skilt- og Punktobjektene som skal vises på skjermen.

For hver enkelt avstandsforespørsel sender NVDB_DataProvider denne videre til NVDB_RESTkit. NVDB_RESTkit sender svaret tilbake til NVDB_DataProvider, som igjen sender svaret tilbake til VegObjektKontroller. Denne klassen formaterer svaret og sender det til HovedskjermViewController.

HovedskjermViewController mottar en NSDictionary med alle vegobjektene som skal vises på skjermen. Klassen tegner ut informasjonen på skjermen og stopper opp igjen. Med jevne mellomrom mottar den deretter avstander som skrives ut på skjermen etter hvert som de mottas.

Dette er gangen i applikasjonen når alt går bra. Det hele repeteres omtrent hvert andre sekund. Et sekvensdiagram (figur 15) viser hvordan klassene kommuniserer. Etter alle forespørsler mot venstre i diagrammet stopper klassene opp og gjør ikke noe mer før en delegatmetode kalles med svar.



Figur 17: Aktivitetsdiagrammet viser hva NVDB_DataProvider gjør når den mottar en forespørsel etter vegobjekter.

5 Grafisk brukergrensesnitt

Flere faktorer var viktige med tanke på det grafiske brukergrensesnittet. Vi skulle følge de syv designprinsippene[53], og i tillegg ta hensyn til Apples retningslinjer for design på iOS[54]. Det var også viktig å tenke på brukervennlighet. Vi skulle følge de fem E'ene[55] og i tillegg tenke på trafikksikkerhet under bruk.

5.1 De syv designprinsippene

De syv designprinsippene er structure, simplicity, consistency, tolerance, visibility, affordance og feedback. De er prinsipper for hva som er god design.

- **Structure** refererer til strukturen og organiseringen av innholdet
- **Simplicity** dreier seg om at informasjonen som vises er tydelig og relevant
- **Consistency** går på at designet er gjennomgående i applikasjonen
- **Tolerance** er toleransen for, og konsekvenser av brukerfeil
- **Visibility** dreier seg om hvorvidt det er opplagt hva de forskjellige funksjonene gjør
- **Affordance** dreier seg om hvorvidt det er opplagt hvordan man bruker funksjonaliteten
- **Feedback** refererer til hvorvidt brukeren får passende tilbakemelding på det han/hun gjør

Siden applikasjonen består av ett skjermbilde med skiltplater som vises til brukeren, og i utgangspunktet ingen krav om interaksjon, oppfylles de fleste av disse prinsippene automatisk. Innholdet er organisert på en strukturert måte, det vises ingen unødvendig informasjon, designet er gjennomgående, og funksjonaliteten og bruken er selvforklarende.

Det er få muligheter for brukeren til å gjøre feil i applikasjonen. Hvis posisjonstjenesten ikke klarer å levere en posisjon får brukeren beskjed om dette, med et hint om at applikasjonen trenger tilgang til denne tjenesten. Denne feilen kan riktignok også oppstå hvis brukeren f.eks. befinner seg i en tunnel. Det kreves interaksjon når denne feilen inntreffer (brukeren må trykke på "OK"), men vi har vurdert det til at denne feilen sjeldent forekommer. Skjer det en feil med Core Data, NVDB APIet eller liknende vil brukeren få en melding om at applikasjonen ikke fant noe data. Applikasjonen vil prøve igjen etter ca. to sekunder. Meldingen forsvinner så snart applikasjonen klarer å hente data. Les mer om feilhåndteringen i Testdokumentasjonen.

[53] Structure, simplicity, consistency, tolerance, visibility, affordance og feedback (Stone, Jarret, Woodroffe, Minocha, «User Interface Design and Evaluation», 2005, Elsevier Inc.).
[54] https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html.
[55] Prinsipper for brukervennlighet: effective, efficient, engaging, error tolerant og easy to learn (Stone, Jarret, Woodroffe, Minocha, «User Interface Design and Evaluation», 2005, Elsevier Inc.).

Figur 18: Denne feilmeldingen dukker opp når posisjonstjenesten ikke klarer å finne enhetens posisjon.

Figur 19: Denne feilmeldingen dukker opp når det skjer en feil i NVDB APIet eller Core Data.

5.2 Apples retningslinjer for design på iOS

Apples retningslinjer for design på iOS er tilgjengelig på https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html. De går i korthet ut på at man må tenke på hvilke tjenester man tilbyr og til hvilken målgruppe, og tilpasse designet deretter. Designet er viktig, og man må legge vekt på detaljer. Man må også, når man benytter telefonens tjenester og teknologier, bruke dem på en måte som brukere forventer i iOS. Det er også et krav om at applikasjonen må ha unike ikoner og grafikk som skal vises i App Store[56].

I Kjørehjelperen bruker vi trafikkskilt til å vise informasjon. Siden målgruppen er sjåfører og informasjonen som vises dreier seg om veg og trafikk føler vi at dette er et riktig valg. Vi har også designet et eget ikon til applikasjonen som vises på telefonens startskjerm og i App Store.

5.3 De fem E'ene

De fem E'ene er prinsipper for god brukervennlighet. E'ene står for effective, efficient, engaging, error tolerant og easy to learn.

[56] App Store er Apples digitale butikk for applikasjoner og spill.

- **Effective** dreier seg om hvorvidt applikasjonen lar brukeren foreta seg de handlingene han/hun har som mål å utføre
- **Efficient** går på hastigheten og nøyaktigheten utførelsen av handlinger kan gjøres
- **Engaging** dreier seg om hvor harmonisk, engasjerende og tilfredsstillende brukergrensesnittet oppleves for brukeren
- **Error tolerant** går på hvor godt designet hindrer feil, og eventuelt hjelper brukeren videre etter en feil
- **Easy to learn** refererer til hvor lett det er å orientere seg og forstå applikasjonens muligheter

Som med de syv designprinsippene tilfredsstilles de fleste av disse kravene automatisk siden applikasjonen kun har ett skjermbilde og det i utgangspunktet ikke kreves noen interaksjon. Hvor harmonisk, engasjerende og tilfredsstillende brukergrensesnittet oppleves for brukeren er i stor grad subjektivt. Det er allikevel forsøkt å designe skjermbildet på en måte som er oversiktlig og passer til innholdet. Skiltene er store og tydelige, og de viktigste skiltene er størst når det vises mange skilt på skjermen. Bakgrunnen er en svak asfalttekstur. Denne skal være med på å forsterke "veg"-tematikken i applikasjonen, samtidig som at den ikke skal være forstyrrende og ta oppmerksomheten bort fra skiltene. Som nevnt under "5.1 De syv designprinsippene" skal ikke applikasjonen stoppe opp ved feil. Brukeren skal få en passende feilmelding, og applikasjonen skal deretter fortsette å kjøre som normalt.

Figur 20: Ikonet til Kjørehjelperen. Dette vises på telefonens startskjerm og i App Store.

5.4 Trafikksikkerhet

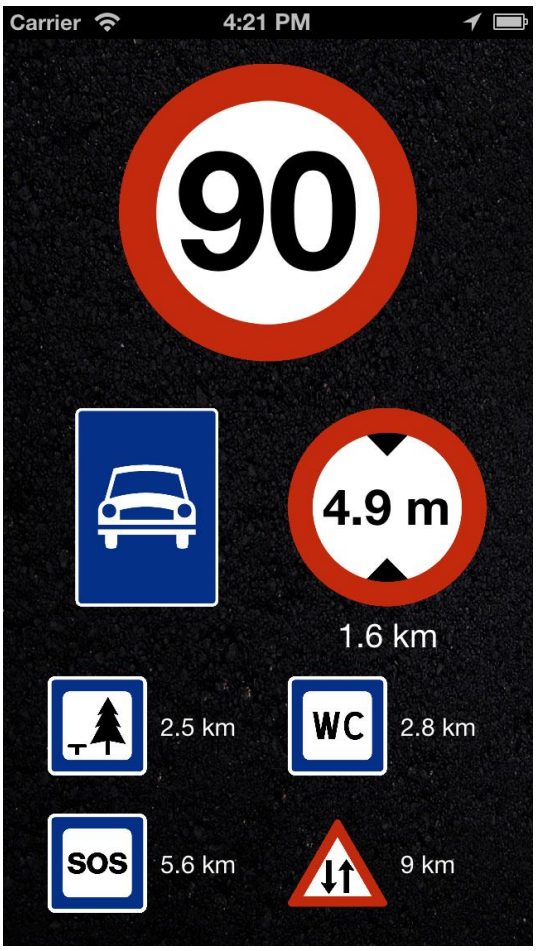
De nye reglene[57] for bruk av mobiltelefon i bil slår fast at mobiltelefonen ikke bare må være fastmontert, men også at det eneste du kan foreta deg under kjøring er å ta samtaler ut og inn og legge på. Applikasjonen er derfor designet slik at du starter den mens du står stille, og deretter skal den ikke kreve noen interaksjon. Unntaket er som nevnt en feilmelding som må klikkes "OK" på når telefonen ikke klarer å finne en posisjon, men dette skal i liten grad skje.

[57] De nye reglene tredde i kraft 01.05.2013.

5.5 Skjermbildet i Kjørehjelperen

Skjermbildet i Kjørehjelperen har en svak asfalttekstur som bakgrunn, og skiltplater med tilhørende tekst under eller ved siden av. De viktigste skiltplatene vises størst, mens når det blir mange skilt samtidig blir de påfølgende skiltene mindre.

Figur 21: Et skjermbilde av applikasjonen i bruk. På en 3,5" skjerm er det kun plass til fire skilt når HUD-knappen vises.



Figur 22: Dette skjermbildet viser den samme posisjonen som figur 20, men på en 4" iPhone 5. Her er også HUD-knappen skult, så her er det plass til syv skilt.

HUD-knappen vises på skjermen for å gjøre bytte mellom HUD-modus og vanlig modus raskt. Hvis brukeren ikke benytter seg av HUD-modusen lar knappen seg skjule i brukerinnstillingene. Dette gir plass til et ekstra skilt. Eier brukeren en iPhone 5 er det i tillegg plass til to ekstra skilt. Dette er fordi iPhone 5 er 4", mot de tidligere modellenes 3,5", og er derfor litt lengre.

I landskapsmodus er det like mange skiltplasser som i portrettmodus. De er riktignok plassert litt annerledes for å få plass til teksten som viser avstanden til objektet.

Figur 23: En 3,5" skjerm i landskapsmodus med HUD-knappen synlig.

Figur 24: En 4" skjerm i landskapsmodus med HUD-knappen skjult.

Når HUD-modus aktiveres tvinges telefonen over i landskapsmodus. Deretter speiles all informasjonen (bortsett fra HUD-knappen) slik at den fremstår riktig når den reflekteres i frontruten.

Figur 25: En 3,5" skjerm i HUD-modus.

Innstillingsskjermen til applikasjonen følger standarddesignet til iOS. Når man benytter brukerinnstillinger i en applikasjon blir disse plassert sammen med innstillinger for andre applikasjoner på telefonen, og får samme utforming som disse.

Figur 26: Innstillingene til Kjørehjelperen ligger sammen med innstillingene til andre applikasjoner på telefonen.

Figur 27: Et utdrag av innstillingene til Kjørehjelperen.

6 Gjenstående arbeid

Selv om Kjørehjelperen er tilnærmet ferdig er det noen ting som gjenstår. NVDB APIet som benyttes er ikke lansert enda. Når dette lanseres er ikke kjent, men det vil trolig skje i løpet av noen få måneder. Inntil dette lanseres kan heller ikke Kjørehjelperen lanseres i App Store.

Før lansering ønsker vi også å gjøre størrelsen på skiltplatene mer dynamiske. Vi ønsker at alle skiltene skal være like store, og at størrelsen reguleres etter hvor mange skilt som vises på skjermen. Dagens løsning fungerer, men kan oppfattes litt rotete, spesielt i landskapsmodus.

Det vil også foretas flere tester av applikasjonen for å finne feil i både den og NVDB APIet. Feil i applikasjonen er alvorlig, ikke bare kan dette gi et negativt syn på Kjørehjelperen og i verste fall Statens vegvesen, men det kan også være et spørsmål om trafikksikkerhet. Selv om applikasjonen frasier seg ansvar, kan det være meget uheldig om den skulle vise uriktig informasjon om en vegstrekning.