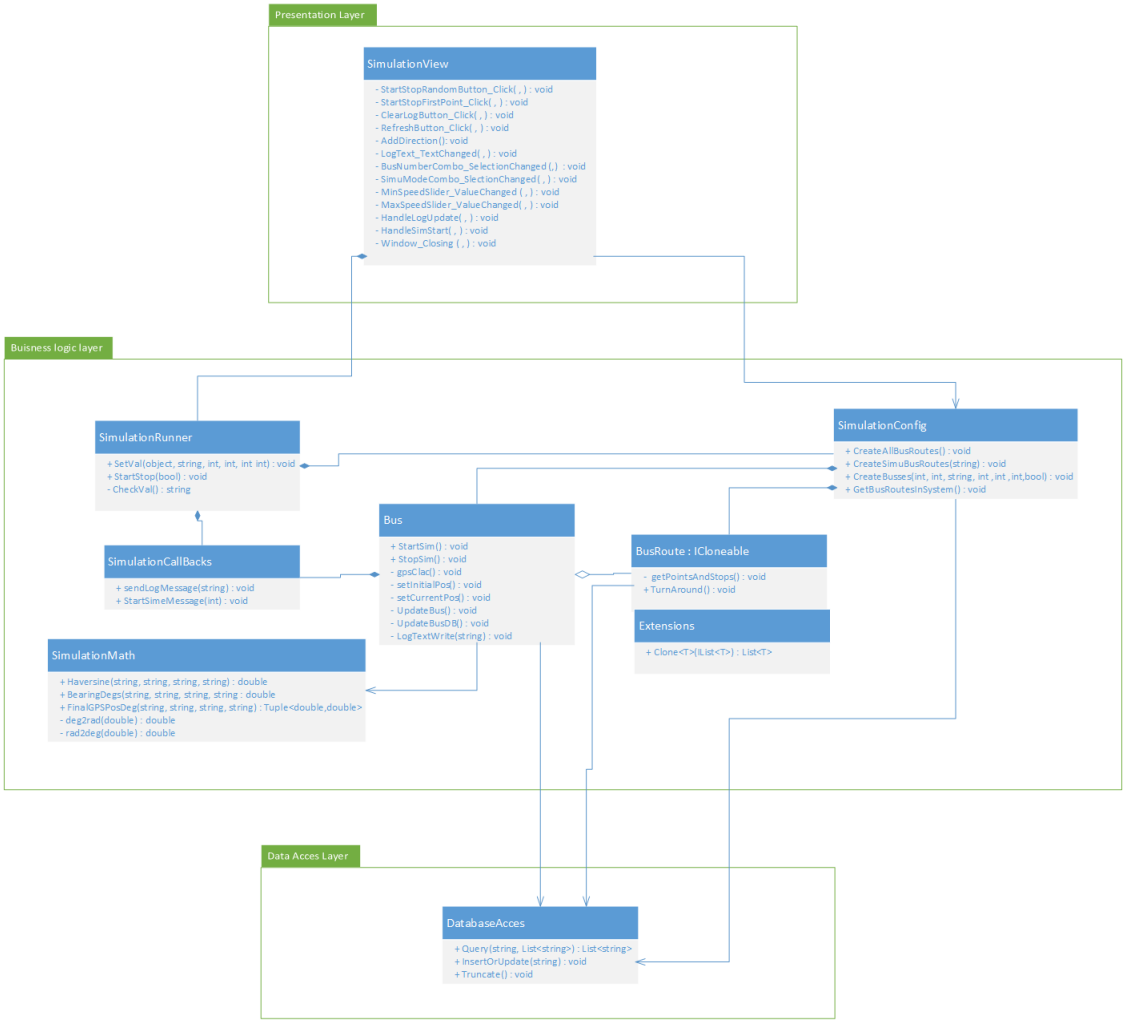
0.0.1 Komponent 4: Simulator

Da det ikke var muligt at kunne tilgå reel positions data for en bus, og implementerings processen ville komplikeres uden, blev det set som gavnligt at bygge en simulator som kunne simulere opførselen af en til flere busser. Denne komponent har til formål at beskrive denne simulator og dens funktioner.

Specifikationer

Simulatoren er bygget som en WPF applikation, hvor .NET framework version 4.5 er blevet brugt. Den er opbygget efter 3-lags modellen. Hertil er der designet en række klasser til at håndtere logik-laget, samt en til at håndtere data-tilgangs laget. Præsententationslaget består udelukkende af ét view. Heri sørges der for, at events bliver håndteret samt påhængtet og afhængtet, hvis det er nødvendigt. På figur **??** ses et klasse diagram, hvorpå det kan ses hvordan simulatoren er bygget op. Klasse diagrammet viser ingen attributer, da de væsentligste kan ses som værende relationerne. De steder hvor der er en association relation, betyder det, at klassen enten tilgår en statisk funktion eller statisk attribut i den associerede klasse.



Figur 1: Klassediagram for simulator. Opbygget som trelags-model

Her følger en kort beskrivelse af hver klasse:

- SimulationView
  - Denne klasse består udelukkende af event-handlers, eller hjælpefunktioner dertil. Den opretter en SimulatorRunner, som igangsættes ved et tryk på en af de to start-knapper. Udover de to start knapper, er der lavet to event handlers, som bruges når der fra logik-laget skal ske ændringer i viewet. Dette sker i sammenhæng med, at en en log-besked skal tilføjes, eller simulatoren igangsættes og viewet skal ændres til en startet/stoppet tilstand.
- SimulationRunner

  - Denne klasse sørger for at starte og stoppe simulatoren. Når simulatoren startes sørger klassen for at simulatoren opsættes. Der undersøges om opsætningen er valid, og annulerer opstarten på simulatoren, hvis de ikke er. Denne klasse er den eneste i systemet der instantiere SimulationConfig, da det er muligt at hente ruter og busser statisk, men ikke oprette dem. SimulationRunner sørger derfor også for ruter og busser bliver oprettet.

- SimulationConfig
  - Denne klasse er bindeledet mellem simulations enhederne, altså alt information om busser og ruter. Hvis klassen er instantieret, kan busruter og busser oprettes, men hvis ikke, kan de stadig tilgås statisk. Der eksisterer to lister af busruter; En der kun indeholder den valgte busrute, og en anden der indeholder alle busruter i systemet, der har busser koblet på sig.

- SimulationMath
  - Klassen bruges i sammenhæng med udregning af ny position for en simuleret bus. Hver Bus instantiere denne klasse ved oprettelse.

- SimulationCallbacks
  - Klassen indeholder events til at håndtere viewændringer fra buisness logic. Dette gøres igennem custom events som hægtes på viewet ved oprettelse. Der er kun to klasser der bruger callbacks; SimulationRunner og Bus.

- BusRoute
  - Denne klasse er repræsentationen af en busrute, i simulatoren. Ved oprettelse sørges der for at rutens punkter hentes fra databasen. I denne sammenhæng hentes punkterne for stoppestederne ikke, da disse ikke relevante for simulatoren. Navnene på rutens stoppesteder hentes dog ud. Navnene tages kun i brug når ruten skal vende. Dette er kun relevant når ruten er kompleks, og det skal undersøges hvilken subrute bussen nu skal køres på. Når bussen vender, vendes hele ruten. Dette betyder også at hver bus skal have sin egen instans af ruten den kører på, og til det formål implementerer BusRoute IClonable, og derfor

indeholder den funktionen Clone(). Denne returner en ny instans, men en kopi af den givne rute. Dette bruges i sammenhæng med Extensions klassen.

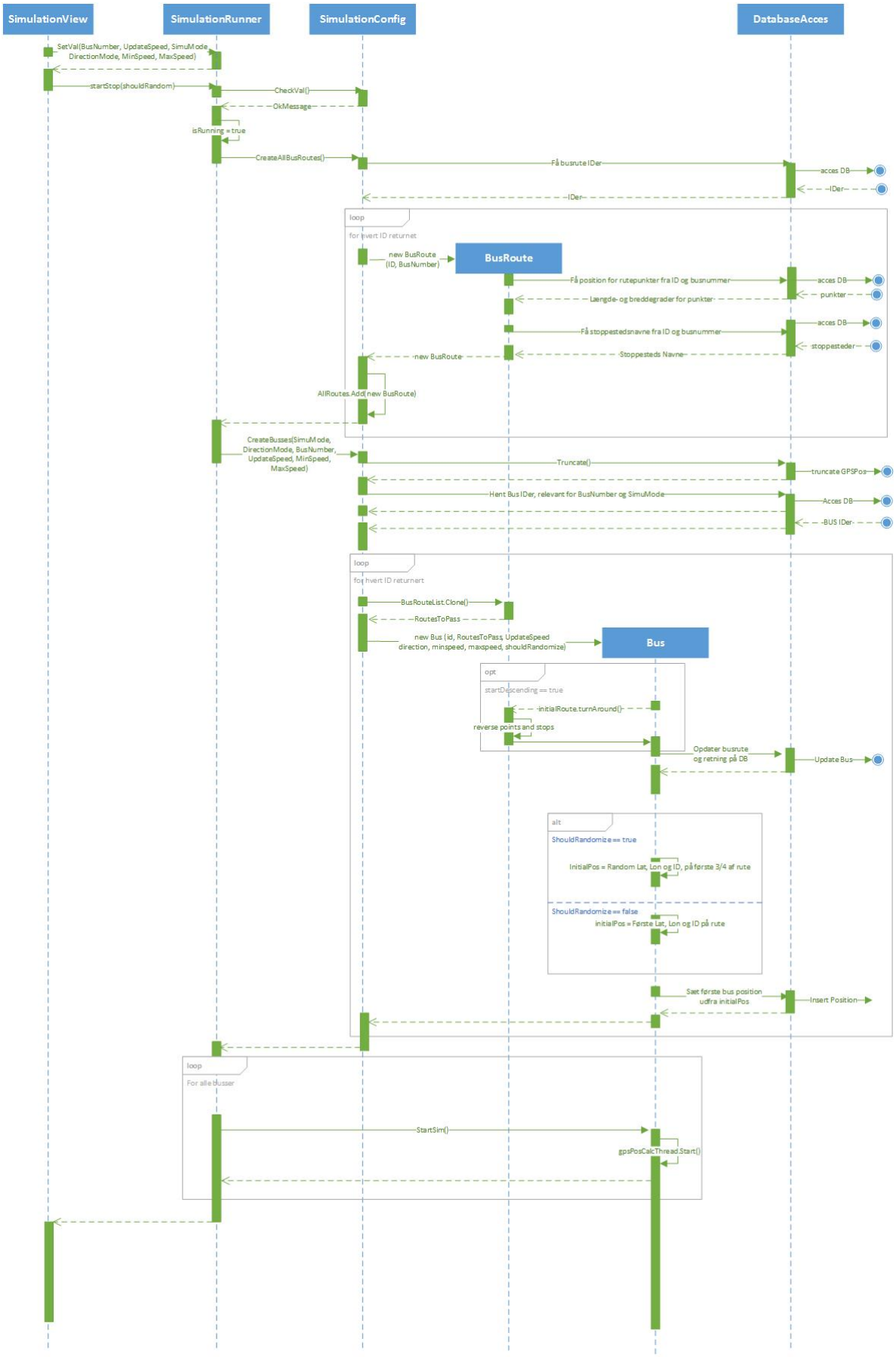
- Extension
  - Når en busrute skal kopieres, bruges metoden Clone(), men hvis det er en række af busruter der skal kopieres, blev det set som nyttigt at oprette en extension metode til en liste, med formål at returner en ny liste, som er en klonet kopi af den gamle. Denne funktion kan dog kun bruges, hvis elementerne i listen implementerer IClonable, som i dette tilfælde er BusRoute objekter.
- Bus
  - Klassen er den simulerede version af en fysisk bus, og indeholder derfor alle funktionaliteter en bus har, for at kunne køre på en rute. Det er også i denne klasse den væsentligste simulering finder sted, i det denne klasse indeholder funktionen til at bestemme nye koordinater for bussen. Når en bus bliver oprettet, oprettes der samtidig en tråd til denne. Det vil sige, at hver bus har sin egen tråd, og kører derfor uafhængigt af resten af systemet, med den undtagelser af, at den kaster et event til at logge en besked på viewet. Selvom flere busser kan kører på samme rute, har bussen kun en kopi af ruten, hvilket betyder, at nå en bus vender, vendes hele kopi-ruten. Den tager desuden også beslutningen, om hvilken rute der skal køres på, hvis ruten er kompleks.
- DatabaseAcces
  - Indeholder funktioner til at tilgå datbasen. Mere information om denne klasse kan findes under afsnit *9.2.2: Implementering af persistens i simulator*.

Processer

Der eksisterer to komplekse processer i simulatoren, som udgør hovedfunktionaliteten; Start af simulering, samt udregning af ny position til bus. Disse vil nu beskrives

På figur **??** kan et sekvensdiagram, der beskrives handlingsforløbet ved start af simulering, ses. Handlings forløbet starter når en af de to start-knapper trykkes, og fuldføres

når hver oprettet bustråd er startet. Sekvensdiagrammet repræsenterer solskinsscenariet, hvori samtlige konfigurationer er korrekte og der er internettet kan tilgås. Først sættes simulationsværdierne i SimulatorRunner. Disse består alle de brugerdefinerede værdier, der kan sættes på GUIen. Herefter igangsættes startningsproceduren, ved et kald til Start-Stop(). De brugerdefinerede værdier undersøges for validitet (eg. om der er nogen der ikke er sat, valgt eller skrevet korrekt), og simulatoren sættes til at være kørende. Samtlige busruter, der har en bus bliver oprettet, først i gennem SimulatorConfig, hvor rutenumre, IDer hentes, og BusRoute objektet oprettes. Constructoren for BusRoute, sørger for at hente stoppesteder og punkter relevant for den givne busrute hentes, og sættes i objektet. Samtlige busruter gemmes statisk i SimulationConfig. Herefter oprettes alle busser, på baggrund af hvilken måde der simuleres samt, hvilken retning busserne skal køres. Mere information omkring simulerings muligheder kan findes under "CLW: SIMULERINGS MULIGHEDER AFSNIT". Ved oprettelse bestemmes der, hvilken underrute bussen skal køre på, i tilfælde af, at ruten er kompleks. Dette sker tilfældigt, uden hensyn til, hvilken simulerings mulighed er valgt. Herefter sættes rute og retning bussen kører, i databasen. Til sidst findes den første position bussen skal køre på. Hvis bussen skal starte først på ruten, vælges det første punkt fra listen. Hvis ikke, vælges der et tilfældigt punkt på de første tre fjerdedele af ruten. Bussen gemmes herefter i listen af alle busser, og udregningstråden for hver bus startes.



Figur 2: Opdeling af klasse i trelags-model

Formålet med simulatoren er, at en bus kan køre på en rute, uden behov for, at en fysisk bus, skal køre på ruten. På kodeudnit ??,?? og ?? kan processen, der står for udregning af ny position ses. Selve udregnings processen er delt op i to kodeudsnit, hvor det første kodeafsnit omhandler initialisering af udregning samt udregninger på rutepunkter, og andet kodeudsnit omhandler bestemmelse af nyt rutepunkt. Kodeudsnit ?? omhandler busvending samt eventuelt ruteskift.

Kodeudsnit 1: Udregning af ny position del 1.

```
1  while (true)
2  {
3    double nextSpeed =SimulationConfig.rand.Next(minSpeed, maxSpeed+1) ;
4    double travelLengthMeters = speed * (1000d / 3600d) * updateSpeed;
5    double currentLength = 0;
6    double nextLength = 0;
7    double brng;
8    if(indexCounter == -1)
9      indexCounter = initialPosIndex + 1;
10
11   while (currentLength < travelLengthMeters)
12   {
13
14     if(indexCounter == initialRoute.points.Count - 1)
15     {
16       currentPos = new Tuple<double,double>(double.Parse(initialRoute.points[indexCounter].Item1),
           double.Parse(initialRoute.points[indexCounter].Item2));
17       UpdateBus();
18       break;
19     }
20
21     if (currentPos.Item1 != 0 && currentPos.Item2 != 0 && nextLength == 0)
22     {
23       nextLength = sMath.Haversine(currentPos.Item1.ToString(),
24         currentPos.Item2.ToString(),
25         initialRoute.points[indexCounter].Item1,
26         initialRoute.points[indexCounter].Item2);
27
28       brng = sMath.BearingDegs(currentPos.Item1.ToString(),
```

```
29          currentPos.Item2.ToString(),
30          initialRoute.points[indexCounter].Item1,
31          initialRoute.points[indexCounter].Item2);
32      }
33      else
34      {
35        nextLength = sMath.Haversine(
36          initialRoute.points[indexCounter - 1].Item1,
37          initialRoute.points[indexCounter - 1].Item2,
38          initialRoute.points[indexCounter].Item1,
39          initialRoute.points[indexCounter].Item2);
40        brng = sMath.BearingDegs(
41          initialRoute.points[indexCounter - 1].Item1,
42          initialRoute.points[indexCounter - 1].Item2,
43          initialRoute.points[indexCounter].Item1,
44          initialRoute.points[indexCounter].Item2);
45      }
46  ...
```

Det første der sker under udregningen er, at det bestemmes hvor hurtigt bussen skal køre ved denne opdatering. Dette er en tilfældig værdi mellem den satte minimum og maximum hastighed. Denne hastighed bruges til at udregne hvor langt bussen skal køre. Da hastighed er udtrykt ved kilometer i timen, konverteres den til meter i sekundet. Opdaterings hastighedne ganges på, da denne er et udtryk for, hvor lang tid bussen har kørt på den givne hastighed. Hvis dette er første gang opdateringen foregår, sættes indexCounter til det valgte start index plus 1, da dette symbolere det næste rutepunkt som bussen ikke er kørt forbi. Herefter startes udregninger for nyt rutepunkt. While-løkken symboliserer et inkrementerende rutestyke, hvor currentLength er det stykke, der er blevet udregnet og travelLengthMeters er det stykke der skal rejses. Der undersøges om sidste rutepunkt er nået, og hvis det er, sættes bussens position til dette punkt, og ruten vendes. Dette beskrives efter kodeudsnit **??**. Herefter udregnes det næste rutestykke, og hvilken kurs dette rutestykke følger. Udregningen af længden af rutestykket og kursen kan findes i afsnittet *8.2.5: Anvendt matematik*. Hvis der endnu ikke er udregnet et linjestykke, og hvis bussens nuværende position er sat, udregnes næste linjestykke og kurs udfra bussens position, og næste rutepunkt. Hvis ikke udregnes der mellem forrige og næste rutepunkt. Udregningen fortsætter på kodeudsnit **??**.

Kodeudsnit 2: Udregning af ny position del 2.

```
1  ...
2
3        if (nextLength + currentLength > travelLengthMeters)
4        {
5          double missingLength = travelLengthMeters - currentLength↩
               ;
6
7          if (currentPos.Item1 != 0 && currentPos.Item2 != 0 && ↩
               currentLength == 0)
8          {
9            currentPos = sMath.finalGPSPosDeg(
10             currentPos.Item1.ToString(),
11             currentPos.Item2.ToString(),
12             brng, missingLength);
13         }
14         else
15         {
16           currentPos = sMath.finalGPSPosDeg(
17             initialRoute.points[indexCounter - 1].Item1,
18             initialRoute.points[indexCounter - 1].Item2,
19             brng, missingLength);
20         }
21         SetCurrentPos();
22         break ;
23       }
24       else
25       {
26         currentLength += nextLength;
27       }
28       string currPosMsg = "Bus " + bID.ToString() +
29         ", new endpoint reached, index: "
30         + (indexCounter + 1).ToString();
31       LogTextWrite(currPosMsg);
32       indexCounter++;
33     }
34     Thread.Sleep(updateSpeed * 1000);
35   }
36 }
```

Næste del af udregningen starter med, at der undersøges om den nyudregnedelængde sammenlagt den totale udregnede længde, er længere end det stykke, bussen skal køre. Hvis det ikke er, inkrementeres den total udregnede længde med den nye. Hvis det derimod er, skal bussens position være mellem det forrige rutepunkt og det næste. Variablen mis-

singLength indeholder den længde bussen skal køre fra det rutepunkt, med den udregnede kurs. Denne variabel er udregnet som længden bussen skal køre, minus det stykke der allerede er udregnet. Herefter undersøges der, hvilket punkt der skal bruges, som værende det punkt, finalGPSPosDeg skal udregne fra (for mere information, se afsnit *8.2.5 - Anvendt matematik*). Hvis den nuværende position ikke er sat endnu, og bussens næste position er på samme linjestykke som forrige (currentLength == 0), så sættes næste position udfra bussens nuværende. Hvis ikke, udregnes der udfra det forrige rutepunkt.
Når et nyt rutepunkt nås, startes et logging-event, som sender GUIen en ny loggingbesked. Desuden inkrementeres indexCounteren, så det næste rutepunkt er det gældende. Når en opdatering er fuldført, sover tråden i det antal sekunder der er sat i opdaterings hastigheden.

Kodeudsnit 3: Valg af ny rute ved endestation.

```
1 if (routes.Count == 1)
2 {
3   initialRoute.TurnAround();
4   indexCounter = 0;
5 }
6 else
7 {
8   List<BusRoute> possibleRoutes;
9   string atStop = initialRoute.stops[initialRoute.stops.Count - ↩
        1];
10  possibleRoutes = routes.FindAll(
11    R => (R.stops[R.stops.Count - 1] == atStop) ||
12    R.stops[0] == atStop);
13  if (possibleRoutes.Count == 1)
14  {
15    initialRoute.TurnAround();
16  }
17
18  else
19  {
20    possibleRoutes.Remove(initialRoute);
21    if (possibleRoutes.Count == 1)
22      initialRoute = possibleRoutes[0];
23    else
24      initialRoute = possibleRoutes[SimulationConfig.rand.Next(
25        0, possibleRoutes.Count)];
```

```
26      if (initialRoute.stops[0] != oldRouteStop)
27      {
28        initialRoute.TurnAround();
29      }
30
31    }
32    indexCounter = 0;
33 }
34 UpdateBusDB();
```

Når en bus vender, vil der ske en ruteændring. Hvis bussen kører på en simpel rute, (rute med to endestationer) , vil der ikke ske andet, end at listen af punkter i ruten vil blive vendt, således at det punkt bussen holder ved, vil være det første punkt på ruten.
Hvis ruten derimod er kompleks (ruter med mere end to endestationer) skal det undersøges, hvilken subrute, bussen nu skal køre på. Dette gøres ved første at udtage alle subruter på ruten, hvis første eller sidste stoppested, er svarende til det sidste stoppested på den nuværende rute. Hvis der kun kan findes en rute, må denne være den originale, og denne rute vendes bare. Hvis ikke, fjernes den originale rute fra listen af mulige ruter. Der undsøges om der nu kun er en rute, eller om der stadig er flere. Hvis der stadig er flere, vælges en rute på tilfældigt, men hvis ikke, vælges den rute der er tilbage. Herefter undersøges der, om den valgte rutes første stoppested er det samme som det sidste stoppested på den tidligere rute. Hvis ikke vendes den nye rute. Til sidst resettes counteren, og bussen bliver opdateret på databasen, med en retning og ny rute.

Diagrammer i fuld størrelse og fuld kode med kommentarer kan findes på bilags CDen under Diagrammer/System Sekvens Diagrammer, samt Kode/Simulator.