0.0.1 Komponent 4: Simulator

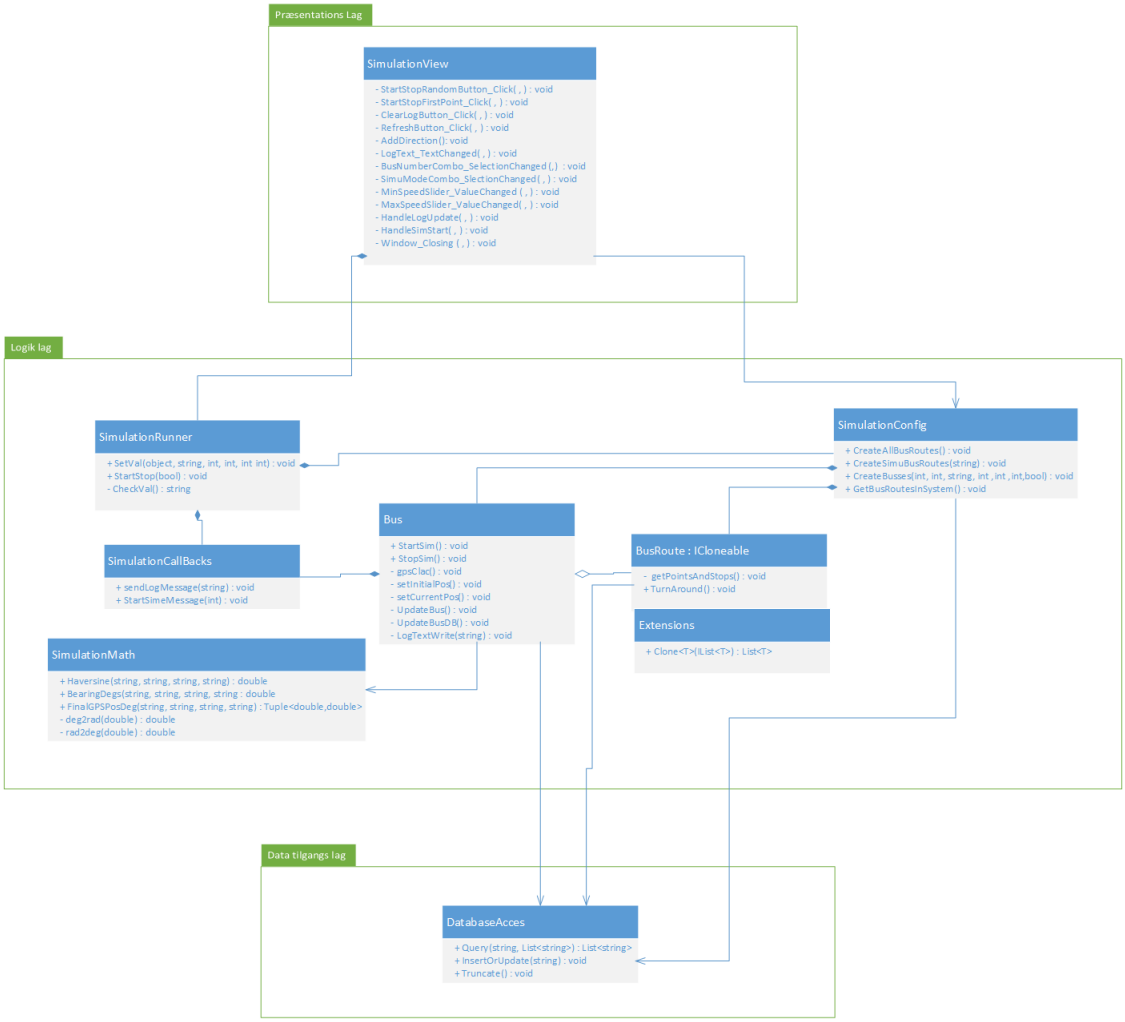
Denne komponent har til formål at beskrive bus simulatoren og dennes funktioner.

Specifikationer

Simulatoren er bygget som en WPF applikation, hvor .NET framework version 4.5 er blevet brugt. Den er opbygget efter tre-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 påhægtet og afhægtet, når det er nødvendigt. På figur 1 ses et klasse diagram, hvorpå det vises, hvordan simulatoren er bygget op. Heri kan klassernes funktioner, samt deres relationer til andre klasser, ses. De steder hvor der er en association relation, betyder det, at klassen enten tilgår en statisk funktion eller statisk attribut i den associerede klasse.

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 opstarts processen annuleres, hvis den 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



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

  - Denne klasse er bindeledet mellem simulations enhederne, altså alt information om busser og ruter. Hvis klassen er instantieret kan relevanter busruter og busser hentes, og sættes i der respektive lister. Listen af kørende busser, og listen af samtlige busruter, som har en bus koblet på sig, kan tilgås statisk.

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

- SimulationCallbacks
  - Klassen indeholder events til at håndtere viewændringer fra logik laget. Dette gøres igennem custom events som hægtes på viewet ved oprettelse. Der er kun to klasser der bruger SimulationCallbacks; 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, da de skal tages 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 indeholder derfor funktionen "Clone". Denne returner en ny instans, men en kopi af den givne rute. Hvis dette ikke gøres, vil det være den samme busrute reference, samtlige busser kører på, hvilket vil betyder, at når én bus vender, vil alle andre busser på samme rute også vende. "Clone"funktionen bruges i sammenhæng med Extensions klassen.

- Extensions
  - Når et BusRoute objekt skal kopieres, bruges metoden "Clone", men hvis det er en liste af BusRoute objekter der skal kopieres, er det nødvendigt at oprette

en extension metode til en liste. Denne metode har til formål at returne 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.

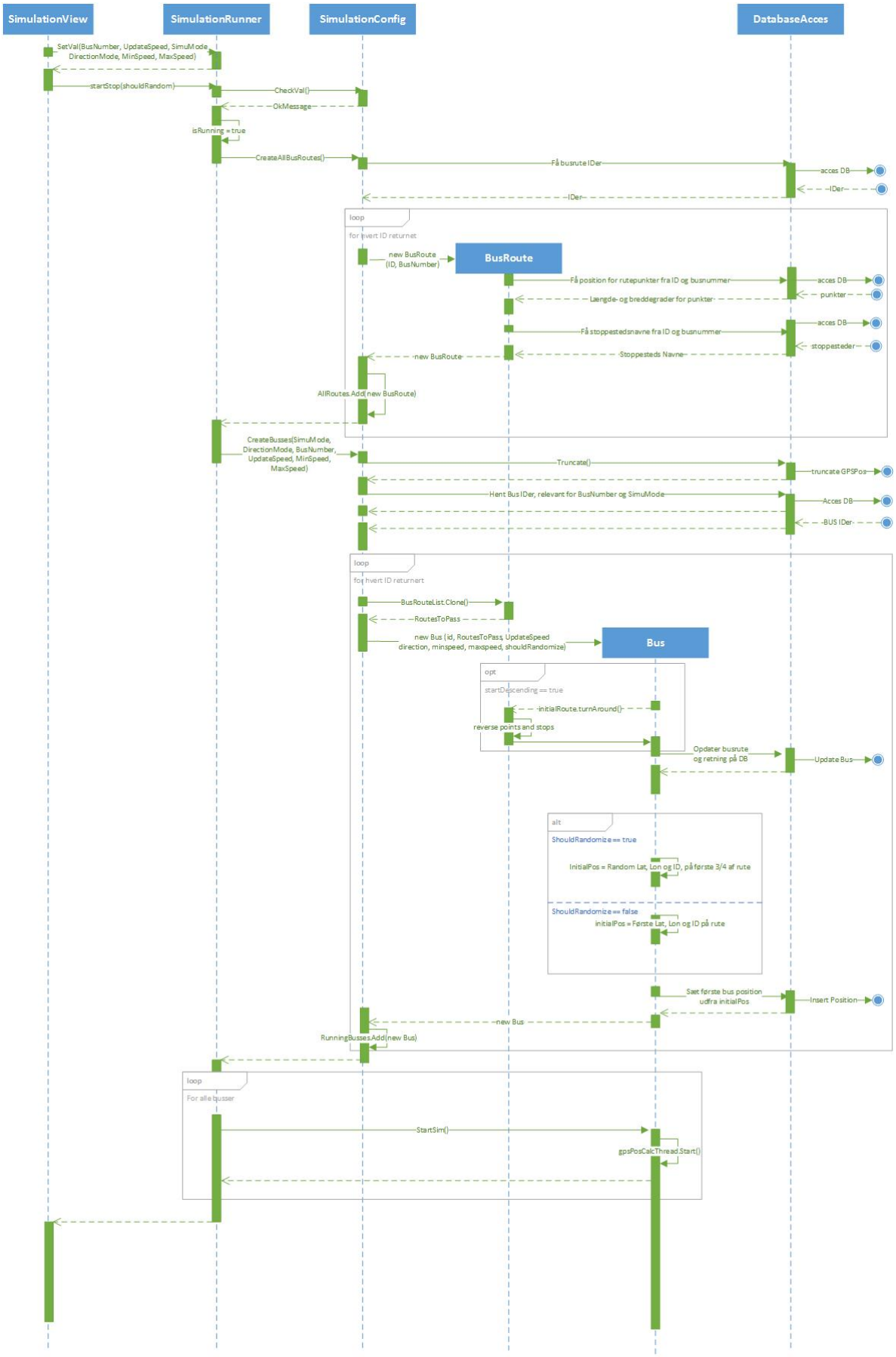
- Bus
  - Klassen er den simulerede version af en fysisk bus, og indeholder derfor alle funktionaliteter en bus har, for at kunne køre på en rute. Det er også i denne klasse den væsentligste simulering finder sted, idet denne klasse indeholder funktionen til at bestemme nye koordinater for bussen. Når et Bus objekt bliver instantieret, 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 logging event igennem SimulationCallbacks klassen. Selvom flere busser kan kører på samme rute, har bussen kun en kopi af ruten, hvilket betyder, at nå én 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*.

Design

Der eksisterer to vigtige implementeringer i simulatoren, som udgør hovedfunktionaliteten; Start af simulering, samt udregning af ny position til en bus.

På figur 2 ses der et sekvensdiagram, der beskrives handlingsforløbet ved start af simulering. 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 internettet kan tilgås. Først sættes simulationsværdierne i SimulatorRunner. Disse består af de brugerdefinerede værdier, der kan sættes på GUIen. Herefter igangsættes startningsproceduren, ved et kald til "StartStop".

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, som har minimum én bus knyttet til sig oprettes. Dette gøres igennem SimulatorConfig, hvor rutenumre og ID'er hentes, og BusRoute objekterne oprettes. Constructoren for BusRoute, sørger for at hente stoppesteder og punkter relevant for den givne busrute og sættes i objektet. Samtlige BusRoute objekter 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 afsnit *3.2: Grænseflader til eksterne system aktører, under afsnittest "Simulator"*. Ved oprettelse bestemmes der, hvilken subrute af den givne busrute, bussen skal køre på. Dette gøres i tilfælde af, at busruten er kompleks. Dette sker tilfældigt, uden hensyn til, hvilken simulerings mulighed er valgt. Herefter sættes ruten og den retning bussen kører, i databasen. Til sidst findes den bussens startposition. Hvis bussen skal starte først på ruten, vælges det første punkt fra listen af punkter i bussens tilknytede BusRoute objekt. Hvis ikke, vælges der et tilfældigt punkt på den første tre fjerdedel af ruten. Dette gøres, så der ikke ved et tilfælde vælges et punkt, der er helt i slutningen af ruten. Bussen gemmes herefter i listen af alle busser, og udregningstråden for hver bus startes.



Figur 2: Sekvensdiagram over start af simuelering

Formålet med simulatoren er, at en bus kan køre på en rute, uden behov en fysisk bus. På kodeudnit 1, 2 og 3 kan processen, der står for udregning af ny position ses. Selve udregnings processen er delt op i to kodeudsnit, hvor det første kodeudsnit omhandler initialisering af udregning samt udregninger på rutepunkter, og andet kodeudsnit omhandler bestemmelse af nyt rutepunkt. Kodeudsnit 3 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 minimums og maksimums hastighed. Denne hastighed bruges til at udregne hvor langt bussen skal køre. Da hastigheden er udtrykt ved kilometer i timen, konverteres den til meter i sekundet, da meter og sekunder er de enheder der arbejdes med. Tiden mellem hver opdatering ganges på, da dette er et udtryk for, hvor lang tid bussen har kørt med den givne hastighed. Hvis dette er første gang opdateringen foregår, sættes indexCounter til det valgte start index plus 1, da dette symboliserer 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 3. 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: Komponent 5: Anvendt matematik* under "Haversine"og "Bearing". 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 det nuværende og næste rutepunkt. Udregningen fortsætter på kodeudsnit 2.

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 nyudregnede længde sammenlagt den totale udregnede længde, er længere end det stykke, bussen skal køre.

Hvis det ikke er, inkrementeres den total udregnede længde med den nyudregnede. Hvis det derimod er, skal bussens nye position være mellem det nuværende rutepunkt og det næste. Variablen missingLength indeholder afstanden fra det nuværende punkt til bussens nye position og er findes som længden bussen skal køre, minus det stykke der allerede er udregnet. Herefter undersøges der, hvilket punkt der skal bruges som initial punktet for udregningen af ny position. Denne udregning kan der læses mere om i afsnittet *8.2.5 - Anvendt matematik*) under "Ny position mellem to punkter"Hvis bussens position tidligere har været sat, og hvis bussens nye position er efter det næste rutepunkt, udregnes bussens nye position udfra bussens forrige. Hvis ikke udregnes den nye position udfra det forrige rutepunkt, relativt til bussens position. Når et nyt rutepunkt nås, startes et logging-event, som sender en ny loggingbesked til GUI'en . 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, 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, skal det undersøges, hvilken subrute, bussen nu skal køre på. Dette gøres ved først at udtage alle subruter på ruten, hvis første eller sidste stoppested, er svarende til den endestation bussen er ved. Hvis der kun kan findes én rute, må denne være den nuværende, og ruten vendes bare. Hvis ikke, fjernes den nuværende rute fra listen af mulige ruter og der undsøges om der nu kun er en rute listen, eller om der stadig er flere. Hvis der stadig er flere, vælges en rute tilfældigt, men hvis ikke, vælges den rute der er tilbage. Herefter undersøges der, om den valgte rutes første stoppested er det samme som den endestation som bussen er ved. Hvis ikke vendes den nye rute. Til sidst sætter indexCounteren til nul, og bussen bliver opdateret på databasen, med en ny retning og rute. Dette sker igennem DatabaseAcces klassen.

Diagrammer i fuld størrelse og fuld kode med kommentarer kan findes på bilags CDen under Diagrammer/System Sekvens Diagrammer, samt Kode/Simulator. Muligheder for simulering beskrives under "CLW: reference til noget"