Projekt Trains

Kurs: **Objektorienterad Programmering i C++**

Student **David Hegardt – dahe1501**

Datum: **2016-10-28**

Utvecklingsmiljö

Utvecklingsmiljön jag har använt är Microsoft Visual Studio 2015. Plattform är Windows 10

Beskrivning av uppgiften

Uppgiften har gått ut på att skapa en simulation av hur ett antal tåg rör sig under ett dygn eller tidsintervall inom detta dygn. Tåglinjerna existerar som abstrakt information, utifrån denna skall tåg försöka att plockas ihop utifrån de vagnar som finns tillgängliga på stationerna. Om tåg ej kan kopplas samman så görs ett nytt försök senare, men avbryts då simulationen tar slut. Implementerade funktioner i programmet :

- Möjlighet att välja start- och sluttid inom dygnet för simuleringen.
- Möjlighet till ändring av intervallängd.
- Möjlighet att välja mellan att stega fram med fast intervallängd eller att låta simuleringen löpa till nästa händelse.
- Välja detaljnivå för informationen som lämnas om fordonen, antingen id+typ eller fullständig information om fordon.
- Möjlighet att för ett visst id få direkt information om var fordonet befinner sig.
- Möjlighet att för ett visst id få en historik över hur detta fordon har förflyttat sig hittills under simuleringen.
- Möjlighet att se tidtabellen samt status för alla tåg, inklusive utskrift av uträknad medelhastighet.

Jag har implementerat samtliga funktioner för betyget **B**, vilket är betyget jag siktar på.

Lösning av uppgiften

Fordonshierarki

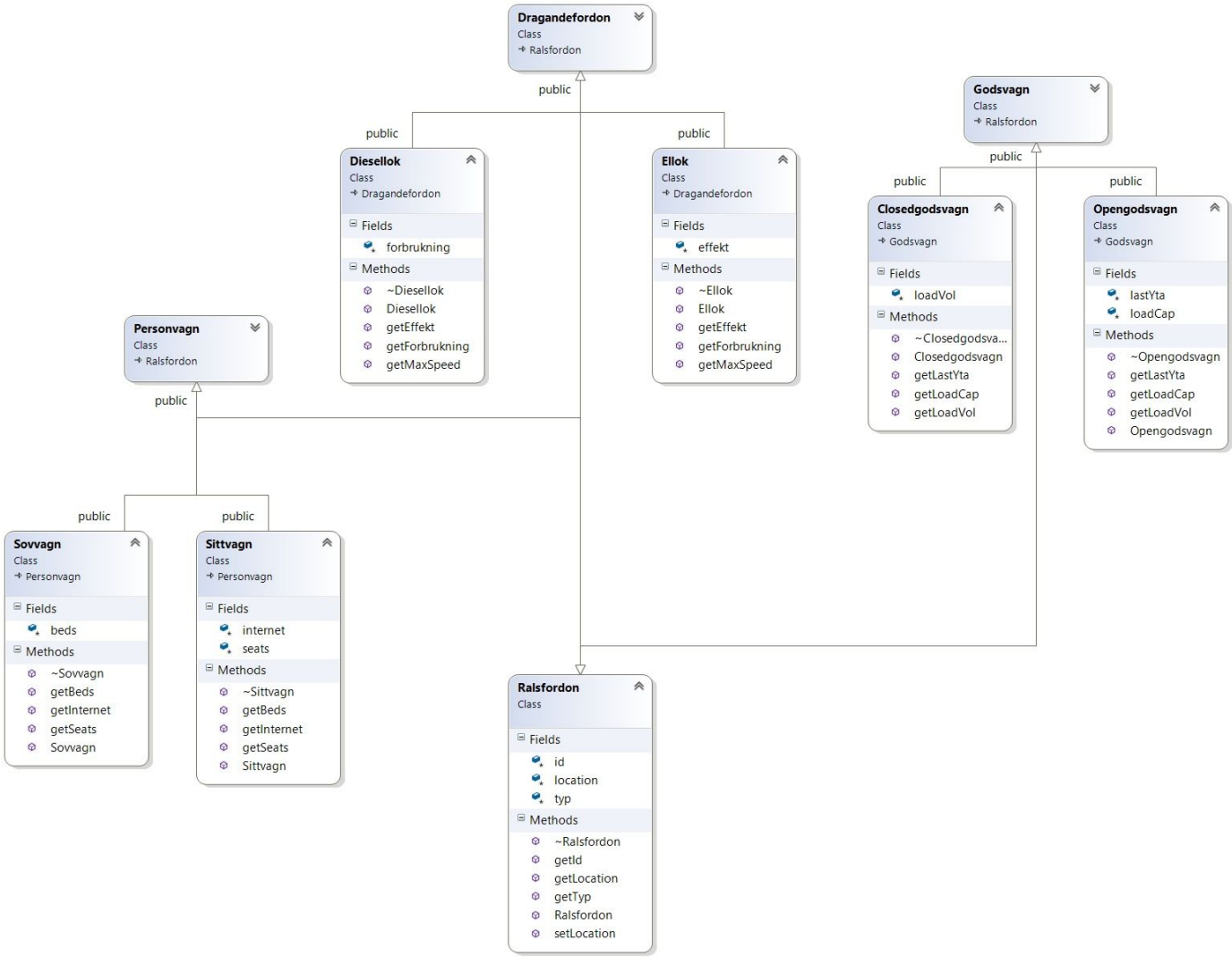
Uppgiften har krävt en viss del objektorienterad analys för att komma fram till vilka klasser som behöver skapas. Hierarkin för fordonen kunde jag rita upp och avgöra relativt snabbt, genom att ta reda på gemensamma parametrar för de ingående klasserna så kunde jag sedan bena ut vilka parametrar som behövs i de olika klasserna. Basklassen kallas här för rälsfordon och innehåller parametrar för id och typ och location. Utifrån detta skapas de deriverade klasserna Personvagn, Dragandefordon samt Godsvagn. Dessa utgör i sin tur basklasser för ytterligare deriverade klasser enligt nedan.

Ralsfordon -> Personvagn -> Sittvagn och Sovvagn

Ralsfordon -> Dragandefordon -> Diesellok och Ellok

Ralsfordon -> Godsvagn -> Closedgodsvagn och Opengodsvagn

Basklassens funktioner är pure virtual då basklassen inte skall instantieras. Vid utskrift används type_cast funktioner för att komma åt medlemsfunktionerna för de deriverade klasserna.


Dragandefordon
Class
Ralsfordon
public
Godsvagn
Class
Ralsfordon
public
public
Diesellok
Class
Dragandefordon
Fields
forbrukning
Methods
~Diesellok
Diesellok
getEffekt
getForbrukning
getMaxSpeed
Ellok
Class
Dragandefordon
Fields
effekt
Methods
~Ellok
Ellok
getEffekt
getForbrukning
getMaxSpeed
public
public
Closedgodsvagn
Class
Godsvagn
Fields
loadVol
Methods
~Closedgodsva...
Closedgodsvagn
getLastYta
getLoadCap
getLoadVol
public
Opengodsvagn
Class
Godsvagn
Fields
lastYta
loadCap
Methods
~Opengodsvagn
getLastYta
getLoadCap
getLoadVol
Opengodsvagn
Personvagn
Class
Ralsfordon
public
public
public
Sovvagn
Class
Personvagn
Fields
beds
Methods
~Sovvagn
getBeds
getInternet
getSeats
Sovvagn
Sittvagn
Class
Personvagn
Fields
internet
seats
Methods
~Sittvagn
getBeds
getInternet
getSeats
Sittvagn
Ralsfordon
Class
Fields
id
location
typ
Methods
~Ralsfordon
getId
getLocation
getTyp
Ralsfordon
setLocation

Inläsning och skapande av objekt

När fordonsklasserna var klara så var det dags för nästa del i projektet, inläsning och skapandet av objekt utifrån klasserna beskrivna ovan. Flödet i exekveringen kräver att alla filer läses in först och objekt skapas utifrån detta för att simulationen sedan ska kunna köras. Jag valde att skapa objekten i samband med att inläsningen sker, tåg samt rälsfordon läses från separata filer. Inläsningen startar i Datafile.cpp som sedan använder sig utav hjälpklasser för att skapa fordon och tåg. Jag skapade en separat klass för hantering av skapandet av tågkartan, i klassen trainmap så använde jag en multimap för att kunna spara informationen och göra den sökbar. Map lämpar sig väldigt bra för detta då jag utnyttjar distansen som nyckel för datan som ska presenteras. Utifrån detta skapade jag sedan en sökfunktion där du kan söka på både distans från A->B och B->A då distansen är densamma. Med hjälp av denna funktion kunde jag utnyttja distansen för att räkna ut medelhastigheten för tågen.

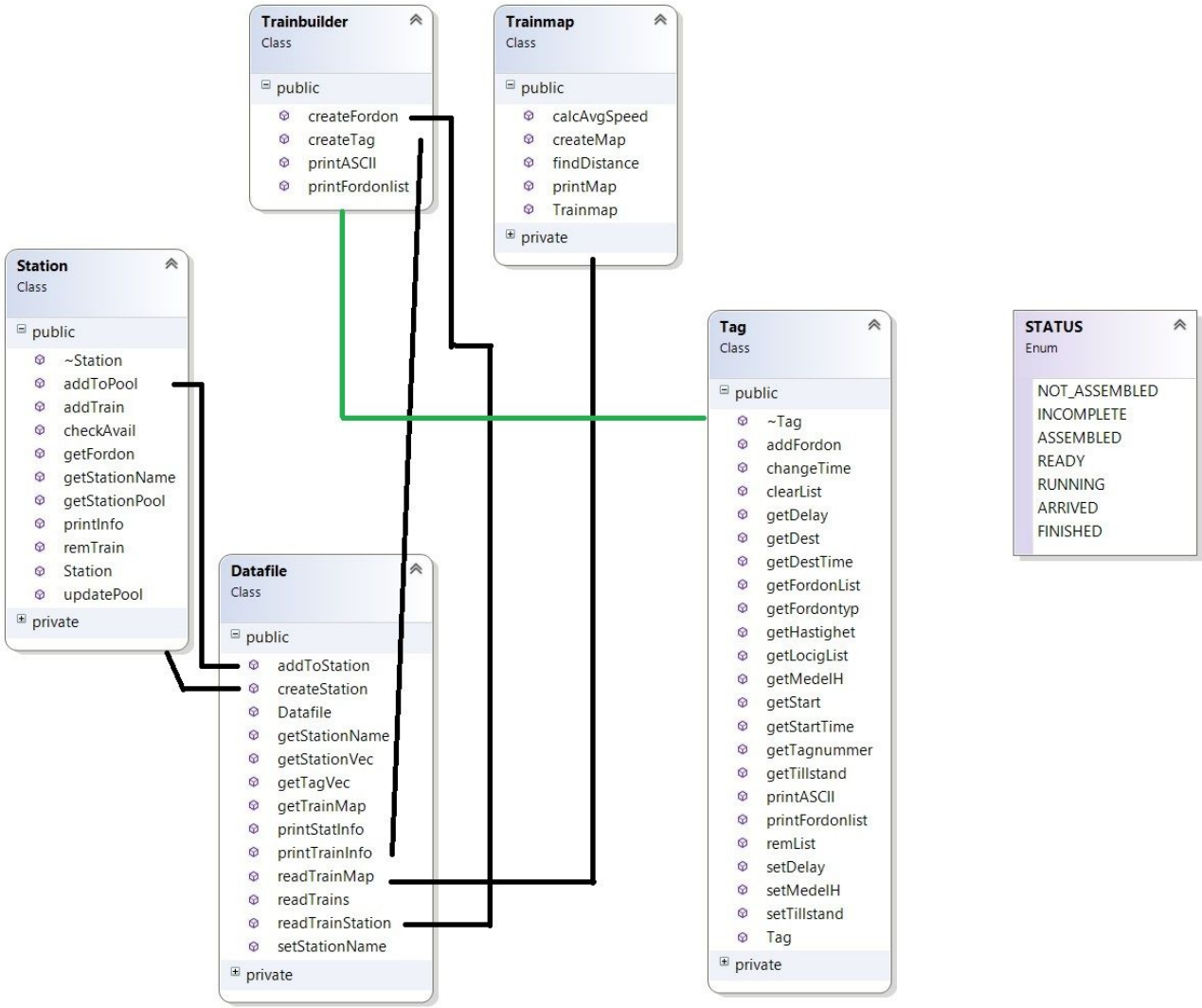
När rälsfordon läses in så har jag valt att lagra dessa som smart_ptrs i en vector, smart_ptrs är lämpligt då objekten kommer att skickas runt under simulationens gång. Vid inläsning av fordon så skrev jag en överlagring för istream operatorn för fordonsobjekt, denna anropar sedan Trainbuilder klassen för att skapa själva fordonet. För skapandet av fordon samt tåg har jag utnyttjat statiska funktioner, likt en fabriksklass så är dessa funktioner skrivna som static för att kunna anropas utan att instantieras. Trainbuilder klassen utnyttjar make_shared för pekaren och returnerar ett nytt fordon beroende på vilka inparametrar som skickas med i överlagringen. För varje stationsinläsning anropas createStation för att skapa stationen innan inläsningen sker av fordonen. Här använder jag också shared_ptr. När ett fordon är inläst så skickas det till funktionen addToStation där nuvarande stationsnamn hämtas upp och läggs till i stationens fordonspool. Fordonspoolen är en vector av shared_ptrs som lagrar fordonen.

På samma sätt sker inläsningen av tåg, genom anrop till Trainbuilder klassen från den överlagrade ifstream operatorn samt en shared_ptr skickas tillbaka. När ett tåg är skapat skickas det till en vector för att sedan kunna utnyttjas under simulationen. När filinläsning sker så sker även felkontroller som kastar felmeddelande om filen inte kan läsas. Den här uppdelningen har lämpat sig bra för exekveringen av programflödet då inläsningen ska ske först i programmet så sker all inläsning härifrån och skapandet av objekt sker i Trainbuilder klassens statiska medlemsfunktioner. På det här sättet var det det enkelt att se ifall inläsningen går som den ska och lättare kunna lösa och isolera problem som hade uppstått ifall man istället hade valt att göra inläsning och skapande i samma klass. Genom att överlagra inläsningsoperatorn så var det enkelt att anropa funktionen för varje fordon som skulle läsas in. Jag har valt att använda shared_ptr för den dynamiska minneshanteringen. Exekveringen sker enligt nedan och förtydligas genom klassdiagrammet hur funktionerna utnyttjar varandra.

Datafile (inläsning sker av trainmap) -> Anropa Trainmap klass -> Skapa karta utifrån medlemsvariabel i Datafile
Datafile (inläsning sker av stations -> Anropa överlagring -> Anropa trainbuilder och

returnera rälsfordon -> Låt Datafile lägga till fordon till station
Datafile (inläsning sker av stations -> Anropa överlagring -> Anropa trainbuilder och returnera tåg-> Låt Datafile lägga till tåg till tåg vektor för att kunna utnyttjas senare.

Datafiles funktioner anropas ifrån klassen Trainstatus.



Eventhantering och Simulation

Nästa del i programmet handlar om eventhanering och körandet av själva simulationen. Utifrån detta har jag byggt upp klassen Trainstatus för händelser som ska inträffa vid olika Event. Upplägget utgår ifrån att tåg befinner sig i de olika tillstånden NOT_ASSEMBLED,INCOMPLETE,ASSEMBLED,READY,RUNNING,ARRIVED,FINISHED. Här har jag valt att använda const enums för att kunna utnyttja i samtliga klasser som använder sig utav Tågklassen.

Utifrån detta har jag även skapat händelser som ska ske i kronologisk ordning, genom att hantera event i en egen klass är det lättare att få överblick över vad som händer vid varje givet event, detta gör det enkelt att isolera problem relaterat till de funktioner som sker vid och på så sätt isolera problem som rör själva simulationen och tidshantering samt de problem som rör händelser som ska ske med själva tåget. Trainstatus utnyttjar funktionerna i Datafile som anropas i kronologiskt ordning, först sker inläsning utav samtliga filer och objekten skapas. Här anropas även funktion för att lagra vilken plats fordonen befinner sig på. Trainstatus utnyttjar och hämtar vektor för stationerna samt tågen som är medlemsvariabler i Trainstatus, här används också shared_ptrs till både tåg och fordon. För lagring utav fordon i tåg används list av shared_ptr. I funktionen tryAssemble sker själva påkopplingen av tåg, här utnyttjas stationerna för att först kolla att rätt fordon är ledigt genom checkAvailible som returnerar en bool om fordon är tillgängligt eller inte, detta säkerställer att fordon inte kan skickas på om de inte är tillgängliga.
Om ett fordon är tillgängligt anropas funktion i Station för att koppla på fordon, fordonen sorteras först efter ID för att fordon med lägst id som matchar typ ska kopplas på. Om fordon finns returneras fordon från stationspoolen och kopplas på rätt tåg i tryAssemble. Om fordon inte finns ändras tågets tillstånd till incomplete. För att ta reda på vilka fordon som behövs använder jag en list av string med vilka typer som ska kopplas på, utifrån detta skickas typ av fordon in i funktionen som en sträng och rätt fordon returneras. Denna list är en medlemsvariabel för varje tåg, och rensas vart eftersom rätt fordon lyckas kopplas på.

När ett fordon inte kopplas på utnyttjas Eventklassen för att göra kontinuerliga nya försök att koppla på fordon. Ready,Running och Arrived ändrar status för tåget och skrev tidigare ut all information om vad som sker, detta flyttade jag senare till Event klasserna som hanterar utskrift och loggning utav händelser. När tåget har kommit fram till stationen ska fordon laddas in till destinationen. I finishup funktionen hämtas rätt station fram, de påkopplade fordonen loopas och läggs till stationens fordonspool, tåget flyttas in till stationens lista av tåg.

Ytterligare funktioner finns för att kunna skapa och hålla reda på vart fordon befinner sig och en historik över detta. För detta syfte använde jag en multimap som lagrar information om plats och typ för varje enskilt fordon. Findfordon söker vid varje anrop igenom samtliga tåg och stationer efter varje fordon, här utnyttjas multimap för att kunna lagra historiken. Utifrån nyckeln id går det att få fram exakt historik för varje fordon.

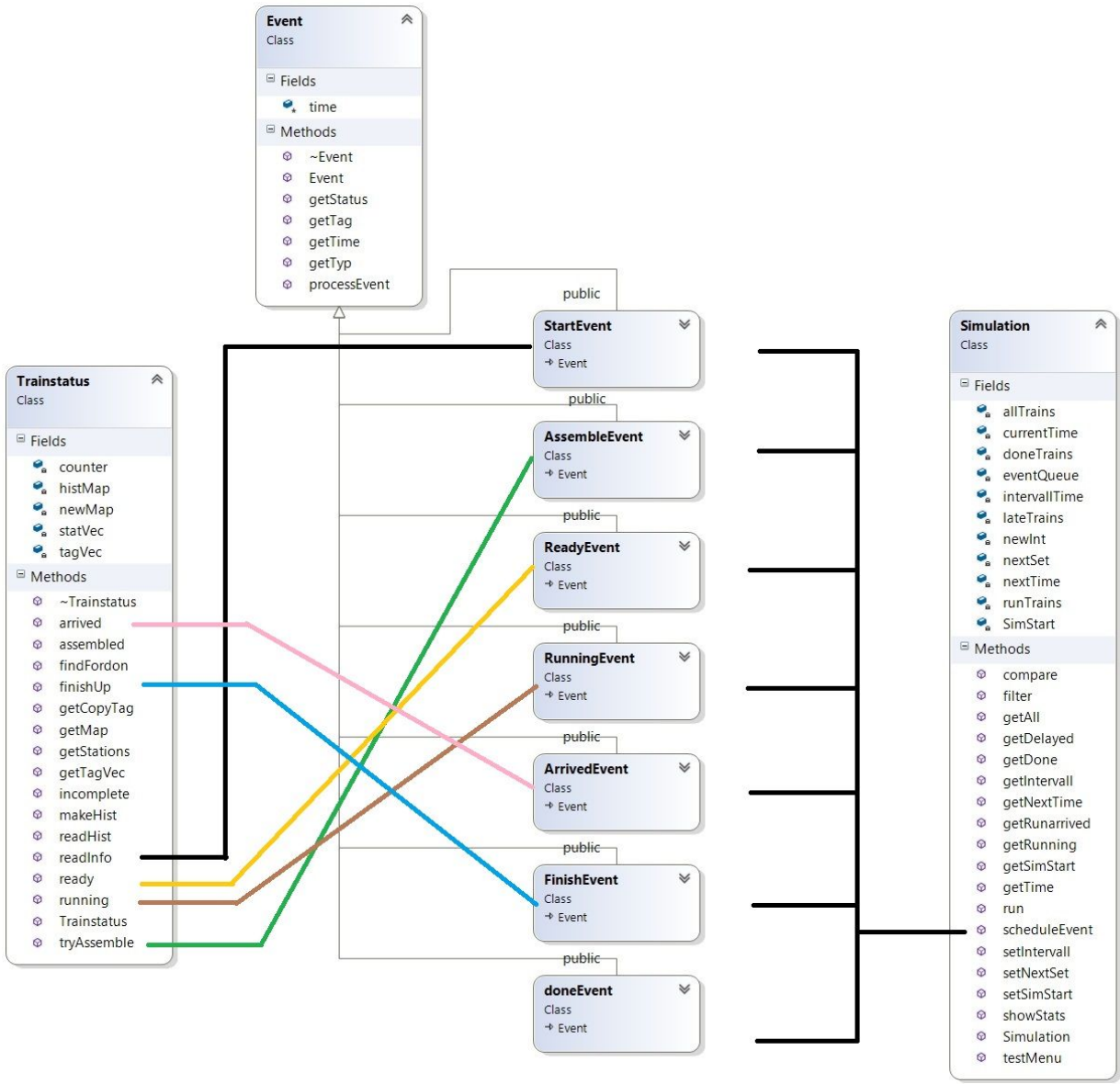
När det gäller Simulation och Eventklasserna har jag utgått från mallen vi fick med om hur en discreet-event-simulation byggs upp. Eventklasserna utgår ifrån basklassen Event och sedan skapas nya klasser för varje Event som skall ske. Jag valde att skapa ett startevent som körs innan simulationen startar, för varje tåg hämtas tid upp för påkoppling av fordon och läggs till en priority_queu som sedan sorterar alla event kronologiskt. Eventen processas sedan ifrån Simulationsklassen och skickar vidare en pekare till tåget för varje event som sker. Här utnyttjas Trainstatus klassens funktioner för hanteringen som ska ske utav tåget. När ett tåg hamnar i incomplete kommer delay tiden för tåget ökas, avg/ank tid ökas på motsvarande förseningen och ett nytt assemble event scheduleras. Detta pågår tills tåget blir komplett eller simulationens tid passerar 23:59. Samtliga tidsökningar utnyttjar const deklarerade tider för att kunna hålla reda på tiden på rätt sätt. En pekare till trainstatus skickas med samtliga event för att kunna utnyttjas för statistik i Simulationsklassen. I Simulation processas eventen utifrån tiden för varje event och en priority queue. Här var det klurigt att få in rätt tid för varje event, då man har möjlighet att processa alla event som sker under en tidslucka eller välja att fortsätta simulationen ett event i taget. Funktionen run utnyttjar bool värden som tar reda på vilken intervall längd som är vald, eller om endast nästa event ska presenteras.

I varje steg av simulationen lagras data för statistik i vektorer som utnyttjar tillstånden från enum const. För att ta reda på när simulationen ska avslutas används räknare som håller reda på vilka tåg som fortfarande kör och vilka som är klara. Här utnyttjas trainstatus ifrån de olika Event klasserna. Anropen illustreras i klassdiagrammet nedan. På detta sätt kan all statistik skrivas ut från simulationen.

Interface klassen hanterar input från användaren och output som ska visas, den innehåller ett menysystem som garanterar att rätt meny visas och ger användaren tillgång till just de funktioner som ska visas vid varje givet tillfälle i simulationen, all inmatning kontrolleras samt att användaren inte har tillgång till funktioner som ska döljas vid olika tillfällen i simuleringen. Här har användaren också möjlighet att välja sluttid och starttid för simulationen. Om användaren väljer att ange en annan starttid för simulationen så körs samtliga event fram till startpunkten, detta är för att få korrekt statistik i simulationen ska visas när den är klar. Ett alternativ till detta hade varit att endast schedulera de event som motsvarar tiden ifrån StartEvent klassen, men som jag förstår det så var det viktiga att få rätt statistik vid slutet av simulationen. Men detta erbjuder även användaren att avsluta simulationen i förtid, då kommer statistik presenteras fram till denna sluttid. Utifrån Interface klassen kan användaren även söka och se historik över varje fordon, skriva ut tidtabell för alla tåg, här presenteras den uträknade medelhastigheten ifall tåget är Running. Möjlighet finns även att söka efter varje fordon och se historik över detta, beskrivet ovan. Samma funktion finns för stationero och tåg, när informationen om tåg skrivs ut finns möjlighet att se en mer detaljerad bild för varje fordon.

För utskrift av fordon används en statisk funktion i trainbuilder, detta var smidigt att göra då man vill kunna skriva ut rälsfordon både från stationer, tåg samt individuellt. Den här funktionen tar hänsyns till vilken loggningsnivå som är vald. För att skriva ut rätt information utnyttjar jag static_pointer_cast som gör så att programmet kommer åt

funktioner i de deriverade klasserna och kan skriva ut respektive värde. Om logglevel 1 är vald kommer endast värdena i basklassen skrivas ut. Loggningsnivå kan ändras från interface klassen för hur mycket information som ska visas. Se nedan för klassdiagram.



Kommentarer och slutsatser

Projektet har varit otroligt intressant och lärorikt, även den objektorienterade analysen var lärorik att testa på och själv få avgöra klasser som behövs. Detta är det hittills mest komplexa program som jag skrivit, däremot har det tagit väldigt mycket längre tid än jag kunde ana, total projektid är långt över 170h. Jag visste att det är den största delen av kursen, men tiden för rättning av buggar och få simulationen att löpa från ett event till nästa/ stega fram från event till nästa var väldigt svår att få till, exemplet som skickades med var bra men det hade varit bra om vi redan tidigt i kursen fått mer inblick i DEDS.
Något som fattades i kursen var lärarledda genomgångar / Q&A sessioner där vi kunde ställa frågor om hur t.ex shared_ptrs ska hanteras i containers och i funktioner.
Genomgången och laborationen av smart pointers var lärorik men jag hade gärna sett mer material där minneshantering gås igenom, vi har aldrig lärt oss debugga applikationer i tidigare kurser och inte heller lärt oss hur man på ett effektivt sätt tar hand om minnesläckor utan det är något vi fått lära oss själva, det kanske skulle ingått i tidigare c++ kurser ?.
Projektet var bra beskrivet och inga större frågetecken behövde redas ut från specifikationen. Bra hjälp och feedback från läraren som hanterade rättningen men kursmaterial / genomgångar var i minsta laget, med mer information och videolektioner så hade möjligtvis mindre tid kunnat läggas på oklarheter kring hur koden ska skrivas.