Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Datateknologi	Vårsemesteret 2024 Åpen
Forfattere: Noa Finkenhagen og Andreas Faye Werner	
Fagansvarlig: Morten Mossige Veileder(e): Tore Fuglestad	
Tittel på bacheloroppgaven: Sikkerhetsvisningskontroll for lakkeringsroboter basert på UNO-rammeverket Engelsk tittel: Safety View Control for paint robots based on the UNO framework	
Studiepoeng: 20	
Emneord: Lakkeringsrobot, ABB, MsAGL, UNO Platform, OPC	Sidetall: 83 + vedlegg/annet: 11 Stavanger 15. mai 2024

Innhold

Sammendrag

Denne bacheloroppgaven hadde som formål å utvikle en applikasjon som grafisk presenterer signalene til sikkerhetstiltak knyttet til en lakkeringsrobot. Et sikkerhetstiltak øker sikkerheten rundt roboten og er et krav ved bruk av industriroboter. Det kan for eksempel være en nødstopp, varsellamper eller fysiske barrierer. Et annet mål var at applikasjonen skulle kjøres på Windows og i nettleseren.

Oppgaven ble utdelt av Universitetet i Stvanger med ABB Bryne som oppdragsgiver.

Formålet med oppgaven ble løst ved å bygge en graf som visualiserer relasjonene mellom sikkerhetstiltakene. Relasjonene mellom sikkerhetstiltakene defineres i en konfigurasjonsfil, som deretter kan lastes opp i applikasjonen. Grafen viser også tilstanden på sikkerhetstiltakene, som for eksempel om nødknappen er trykket inn eller ikke.

Utviklingen av applikasjonen var vellykket til å visualisere sikkerhetstiltakene grafisk. Derimot oppstod det utfordringer med å presentere grafen i nettleseren, grunnet kompatibilitetsproblemer mellom ulike teknologier.

Forord

Denne oppgaven er er skrevet som en del av bachelorgraden i datateknologi ved Universitetet i Stavanger. Oppgaven er et samarbeidsprosjekt med de to undertegnede og ABB Bryne.

Vår takknemelighet går til ABB Bryne for tildelingen av oppgaven, kontorplass og ekspertisehjelp fra Morten Mossige og Tore Fuglestad. Det har vært til stor hjelp under utviklingen av applikasjonen og bacheloroppgaven.

Vi har lært mye om dagens system for samhandling med lakkeringsroboter. I tillegg har det vært en veldig god erfaring å samarbeide med en stor bedrift som ABB og se hvordan de jobber.

Noa Finkenhagen
Andreas Faye Werner
15. mai 2024

Akronymer

ABB	**A**sea **B**rown **B**overi
CIL	**C**ommon **I**ntermediate **L**anguage
CLR	**C**ommon **L**anguage **R**untime
COM	**C**omponent **O**bject **M**odel
CSV	**C**omma-**S**eparated **V**alues
DCOM	**D**istributed **C**omponent **O**bject **M**odel
GB	**g**iga**b**yte
HTML	**H**yper **T**ext **M**arkup **L**anguage
LINQ	**L**anauge **In**tegrated **Q**uery
MAUI	**M**ulti-platform **A**pp **UI**
ms	**m**illi**s**ekund
MsAGL	**M**icro**s**oft **A**utomatic **G**raph **L**ayout
OPC	**O**pen **P**latform **C**ommunications
OPC **UA**	OPC **U**nified **A**rchitecture
SDK	**S**oftware **D**evelopment **K**it
s	**s**ekund
UML	**U**nified **M**odeling **L**anguage
UWP	**U**niversal **W**indows **P**latform
WinUI	**Win**dows **UI** Library
WPF	**W**indows **P**resentation **F**oundation
XAML	**E**xtensible **A**pplication **M**arkup **L**anguage
XML	E**x**tensible **M**arkup **L**anguage

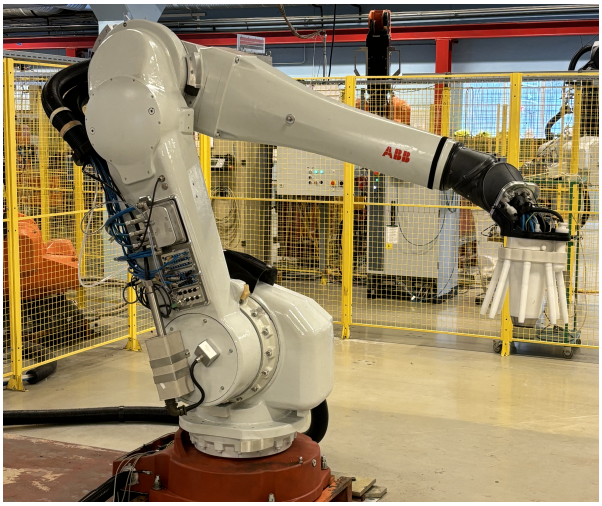
Kapittel 1

Introduksjon

Dette kapittelet tar for seg en introduksjon av oppgaven samt forklaring av målet med applikasjonen som skal utvikles.

1.1 Bakgrunn for oppgaven

Bruk av lakkeringsroboter har funnet veien inn i mange ulike produksjonsindustrier. Ved høyere fart, presisjon og ytelse har lakkeringsroboter blitt en viktig komponent for produksjonen i ulike industrier, som for eksempel i bilindustrien. ABB produserer lakkeringsroboter som blir brukt av andre bedrifter. Avdelingen til ABB på Bryne har fokus på utvikling og testing av lakkeringsroboter, samt programvaren som samhandler med robotene. Figur 1.1 viser et bilde av en lakkeringsrobot ved ABB Bryne.



Figur 1.1: Lakkeringsrobot ved ABB Bryne.

Spesifikt til ABB Bryne sine lakkeringsroboter er det en programvare som er utviklet for intern bruk til å hente informasjon fra robotene. Denne programvaren heter RobView. Denne oppgaven fokuserer på et delprogram i RobView som grafisk presenterer signalene fra sikkerhetstiltakene i roboten.

En hver industrirobot har gitte krav for sikkerhet rundt bruk av roboten. Fra Maskinforordningen (Regulation (EU) 2023/1230) beskrives det ulike sikkerhetstiltak som skal øke sikkerheten ved produksjon og bruk av maskiner. Dette inkluderer også industriroboter. [1] Eksempler på disse sikkerhetstiltakene er blant annet nødstopp-knapper, sikkerhetsgjerder, sensorer og varsellamper. [12]

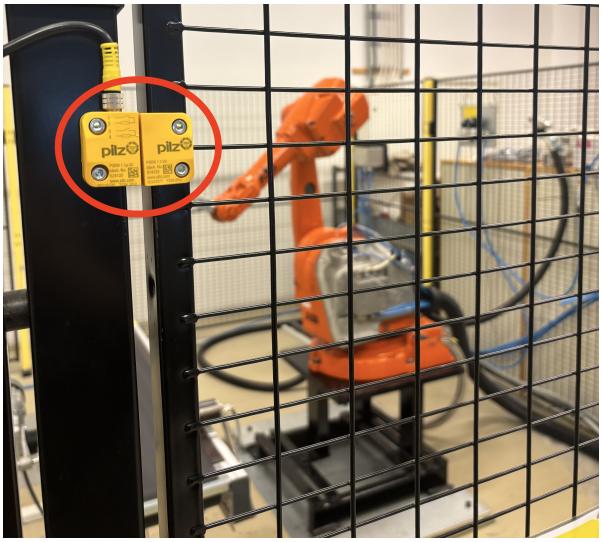
1.1.1 Definisjon på sikkerhetstiltak

I denne oppgaven defineres et sikkerhetstiltak på følgende måte:

Et **sikkerhetstiltak** er en sensor, komponent, knapp eller lignende knyttet til en lakkeringsrobot som forteller om roboten kan starte trygt.

1.1.2 Eksempel på sikkerhetstiltak

Figur 1.2 viser et eksempel på et sikkerhetstiltak på en lakkeringsrobot hos ABB Bryne. Sikkerhetstiltaket er en sensor som er koblet på døren til den fysiske barrieren rundt roboten. Hvis døren er åpen får roboten ikke tilførsel av strøm som igjen vil føre til at roboten ikke starter. Sikkerhetstiltaket sender fra seg et signal som består av enten 1 eller 0 basert på om døren er lukket eller ikke. Det er dette signalet som er grafisk representert i RobView.



Figur 1.2: Sensor tilknyttet dør ved en lakkeringsrobot.

1.1.3 Problemer med dagens system

Problemet med den grafiske visningen av signalene til sikkerhetstiltakene i RobView er at de er hardkodet i programvaren. Dette fører til mye jobb skal man endre på konfigurasjonen av sikkerhetstiltakene. Da må en utvikler med spesialkompetanse endre på kildekoden.

Et annet problem med dagens system er at RobView kun er komptatibelt med Windows. På ABB Bryne er ikke samhandlingen med lakkeringsrobotene kun begrenset til et operativsystem eller enhet. Det brukes både vanlige datamaskiner og håndholdte kontrollere for samhandling med lakkeringsro-

boter. Nettleserkompatibilitet gjør det derfor mulig å kjøre applikasjonen på ulike datamaskiner, siden den ikke er begrenset til et operativsystem. De håndholdte kontrollene bruker Windows-operativsystemet. Figur 1.3 viser et bilde av en håndholdt kontroller på ABB Bryne.

Figur 1.3: Håndholdt kontroller på ABB Bryne.

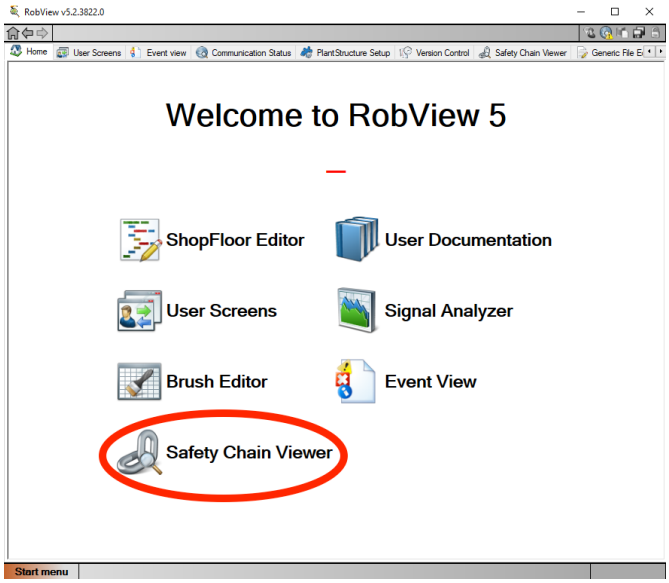
1.2 Om bedriften ABB

ABB (Asea Brown Boveri) er et globalt teknologiselskap som spesialiserer seg innen elektrifisering, prosessautomatisering, motorer og robotteknologi. Selskapet ble grunnlagt i 1883 og har i dag hovedkvarter i Zürich. ABB har en betydningsfull historie innen teknologisk innovasjon og spiller en stor rolle i feltet for automasjonsteknologi. ABB har i dag 105,000 ansatte med en inntekt på over 30 milliarder dollar. [3] [6]

ABB har mange lokasjoner i Norge, med blant annet en avdeling i Bryne. Selskapet har hittil levert over 20,000 lakkeringsroboter over hele verden. Produksjonen av roboter er i dag flyttet fra avdelingen på Bryne, men et ledende utviklingsmiljø er fortsatt til stede. Det foregår omfattende innovasjon og forskning på lakkeringsrobotene til ABB på avdelingen på Bryne. Et eksempel på anerkjent teknologisk utvikling ved ABB Bryne er introduskjonen av verdens første lakkeringsrobot i 1969. [2]

1.3 RobView - Eksisterende programvare på ABB

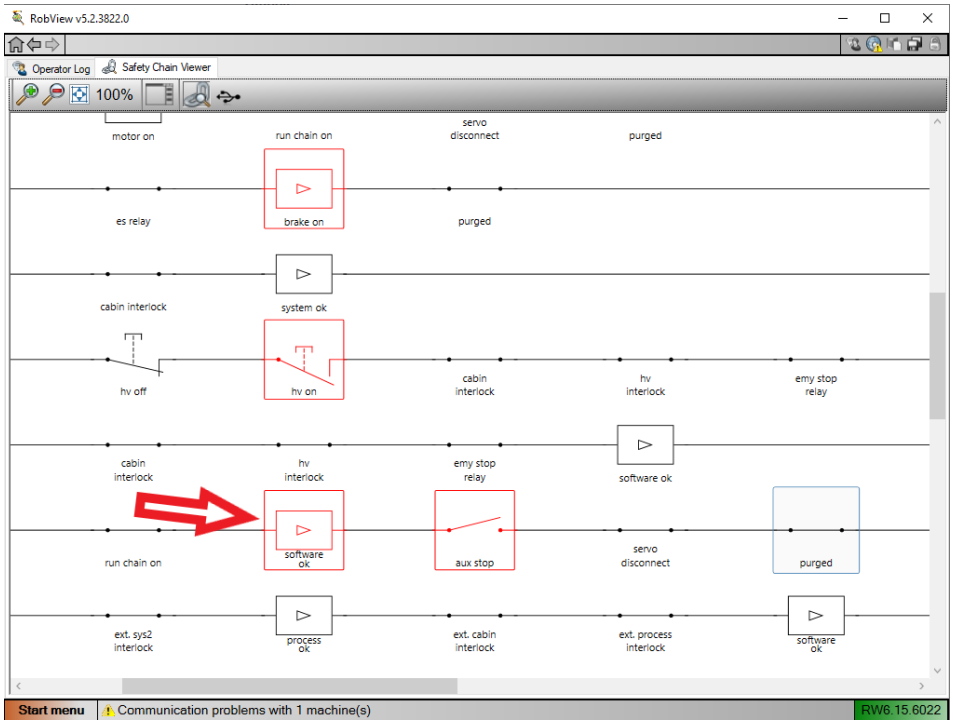
RobView er applikasjonen ABB bruker for å samhandle med sine lakkeringsroboter. Den inneholder flere delprogrammer for å lese ut data om robotene. Et av delprogrammene er Safety Chain Viewer som viser signalene til sikkerhetstiltakene grafisk. Figur 1.4 viser et skjermbilde av RobView sin startside med Safety Chain Viewer markert.



Figur 1.4: Startsiden i RobView med Safety Chain Viewer markert.

1.3.1 Safety Chain Viewer - Grafisk fremstilling av sikkerhetstiltak

Safety Chain Viewer er et delprogram i ABB sin applikasjon RobView. Safety Chain Viewer viser flere vannrette lenker med ulike symboler som representerer forskjellige sikkerhetstiltak. Signalet til sikkerhetstiltaket er representert ved at symbolet til sikkerhetstiltaket enten har fargen rød eller sort. Figur 1.5 viser et utklipp av Safety Chain Viewer og en rød pil som peker på et sikkerhetstiltak med navn *software ok*.



Figur 1.5: Safety Chain Viewer i RobView.

Sikkerhetstiltaket har fargen rød, basert på signalet det sender fra seg. Fargen impliserer at tilstanden til sikkerhetsiltaket ikke er slik det skal være og at roboten derfor ikke kan starte. I dette tilfellet betyr den røde fargen at det er noe galt med en del av programvaren til roboten. Hvis et symbol har fargen sort, betyr det at alt er slik det skal være. Ved bruk av Safety Chain Viewer er det mulig å raskt lokalisere hvilket sikkerhetstiltak som ikke er i riktig tilstand.

1.4 Mål for oppgaven

Målet med oppgaven er å utforske muligheten for utviklingen av en forbedret applikasjon basert på Safety Chain Viewer i RobView. Applikasjonen skal bygge og visualisere en graf basert på en konfigurasjonsfil. Konfigurasjonsfilen representerer relasjonen mellom sikkerhetstiltakene.

Et annet mål for oppgaven er å sjekke muligheten for å utvikle en multiplattform-applikasjon. Spesifikt skal den kjøre på Windows og i nettleseren, ettersom det er plattformene ABB Bryne bruker for å samhandle med lakkeringsrobotene.

1.4.1 Tekniske mål for applikasjonen

Ut ifra gjennomgåtte problemer og gruppens visjon ble det utledet følgende tekniske mål for applikasjonen:

- Grafisk visning av signalene til sikkerhetstiltak, bygget ved opplastning av konfigurasjonsfil
- Applikasjonen skal kunne kjøres som Windows-applikasjon og nettside ved bruk av rammeverket UNO Platform
- Innsending av signaler knyttet til hvert enkelt sikkerhetstiltak
- Avhengig av hvilket signal som blir sendt inn skal fargen på sikkerhetstiltaket endres i grafen
- Direkte kobling til lakkeringsrobot med live data
- Høy ytelse ved riktig utnyttelse av ressursene til datamaskinen som kjører applikasjonen

1.4.2 Brukervennlige mål for applikasjonen

Brukervennlige mål hjelper til med å sette mål for hvordan opplevelsen av applikasjonen skal fremstå for brukeren. Følgende mål er satt for å skape en god opplevelse av applikasjonen for brukeren:

- Mulig å klikke på en spesifikk node i grafen (sikkerhetstiltak) og se informasjon om valgt sikkerhetstiltak
- Siste node i grafen fungerer som en oppsummering av hele kjeden, slik at det blir enklere å feilsøke

- Mulighet for å zoome inn og ut av grafen
- Mulighet for å dra i visningen av grafen
- Egen meny for opplastning av konfigurasjonsfil, informasjon om sikkerhetstiltakene og innsending av signaler

1.4.3 Kvalitetsmål for applikasjonen

Kvalitetsmål innebærer mål som vil øke kvaliteten av programmet. Følgende mål er satt for å sikre god kvalitet i applikasjonen:

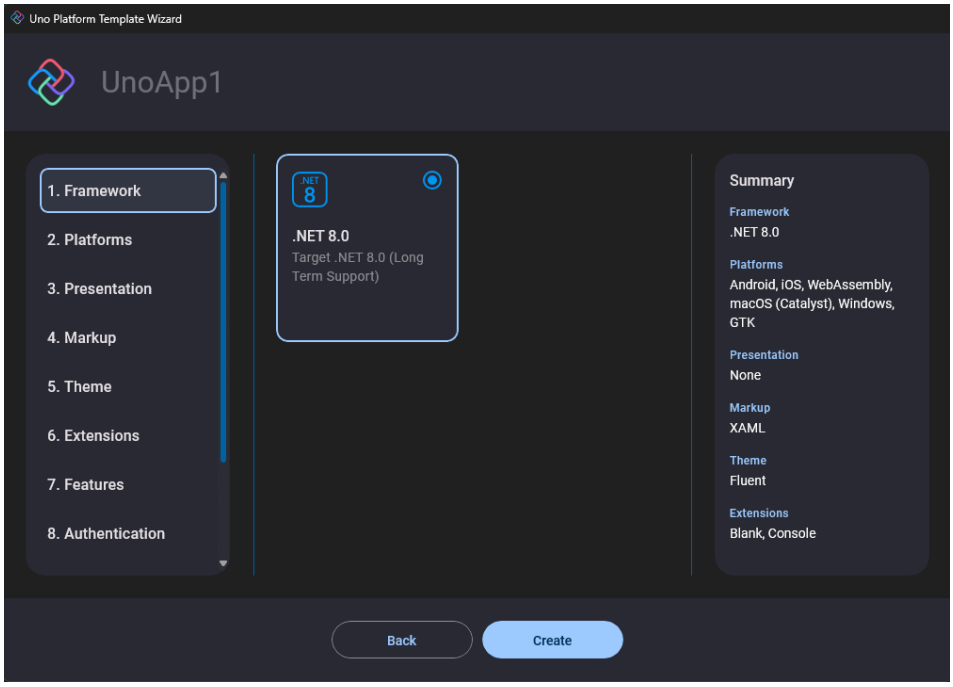
- GitHub automatisk testing ved hver push
- Testdrevet utvikling for å sikre god kvalitet på applikasjonen og riktig funksjonalitet
- Testing av at applikasjonen fungerer på multiplattform

1.5 Utviklingsverktøy

I utviklingen av applikasjonen ble utviklingsmiljøet Visual Studio fra Microsoft brukt. Visual Studio er designet for å utvikle Windows-applikasjoner. I tillegg har Visual Studio flere nyttige funksjoner som debugging, testing og samarbeidsmetoder for enklere utvikling.

Applikasjonen er bygget på UNO Platform-rammeverket. UNO muliggjør utviklingen av multiplattform-applikasjoner. Oppsettet av applikasjonen ble dermed gjennomført ved bruk av UNO sitt konfigurasjonsverktøy. Gjennom UNO sitt konfigurasjonsverktøy er det mulig å velge blant annet plattform, presentasjon og utvidelser. Konfigurasjonsverktøyet blir tilgjengelig når et nytt prosjekt opprettes i Visual Studio. Figur 1.6 viser konfigurasjonsverktøyet til UNO i Visual Studio.

1.5 Utviklingsverktøy



Figur 1.6: Konfigurasjonsverktøyet fra UNO Platform i Visual Studio.

Git er et versjonskontrollsystem som tar hånd om lagring og versjonskontroll av kode. Ved bruk av Git er det mulig for flere å samarbeide på et prosjekt og gjøre endringer i programvaren uten at det påvirker andre som jobber på prosjektet. GitHub ble brukt i dette prosjektet for lagring og vedlikehold av kildekoden. Kildekoden er tilgjenlig her: GitHub - SECView.

Kapittel 2

Teori

I dette kapittelet vil sentrale teknologier og biblioteker som har blitt brukt for utviklingen av applikasjonen forklares og begrunnes.

2.1 Programmeringsspråk og rammeverk for utvikling av applikasjonen

Valget av programmeringsspråket C# var et krav fra ABB Bryne. Grunnen til dette er at RobView er skrevet i dette språket og C# er et av utviklingsspråkene som blir brukt på ABB Bryne. I tillegg har utviklerene på ABB høy kompetanse i C# og .NET miljøet.

2.1.1 C#

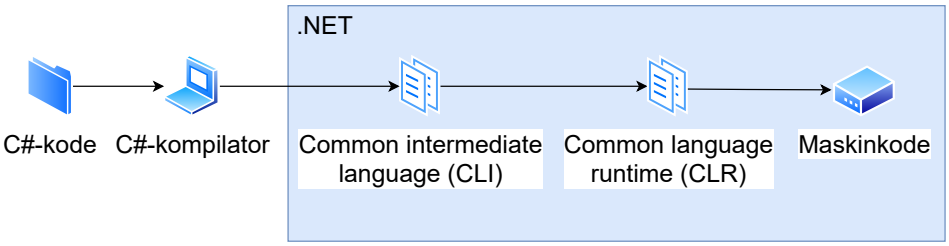
Applikasjonen er bygget på programmeringsspråket C#. C# er et objektorientert programmeringsspråk utviklet av Microsoft som en del av .NET-rammeverket. C# er bygget på C-familien og ligner andre programmeringsspråk som JavaScript, Java og C++. C# er et moderne programmeringsspråk og tilbyr mange teknologier, biblioteker og funksjoner som gjør det enkelt å bygge sikre og robuste applikasjoner. [17]

2.1.2 .NET

.NET er et programvareutviklingsrammeverk utviklet av Microsoft og er designet for å bygge applikasjoner over flere plattformer. .NET er bygget for å utvikle pålitelige, raske og sikre applikasjoner. Hovedkomponentene som .NET er bygget opp av innebærer:

- **Runtime**: Runtime-komponenten er ansvarlig for å utføre kjørekoden for .NET applikasjoner. Komponenten muliggjør ulike funksjonaliteter som minnehåndtering og ressursallokkering under kjøretid
- **Biblioteker**: .NET-biblioteker inneholder ferdiglaget kode som utviklere kan bruke i egne applikasjoner. Dette fører til mye tid og ressurser spart under utviklingsprosessen
- **Kompilator**: Kompilatoren transformerer kildekoden, skrevet i for eksempel C# til maskinkode, som kan kjøres på .NET-runtime
- **SDK**: Software development kit (SDK), gir utviklere alle nødvendige ressurser og verktøy for å utvikle, teste og distribuere programvare for en bestemt platform

Fordelen med .NET er blant annet økt produktivitet på grunn av alle verktøy og biblioteker, sikrere kode, bedre ytelse på grunn av optimaliseringen av .NET og muligheten for .NET-applikasjoner til å kunne kjøres på ulike operativsystemer og enheter. [14] Figur 2.1 viser arkitekturen av .NET og sammenhengen med C#.



Figur 2.1: Arkitekturen av .NET-rammeverket og sammenhengen med C#.

CIL (Common intermediate language) fungerer som en språkpresentasjon

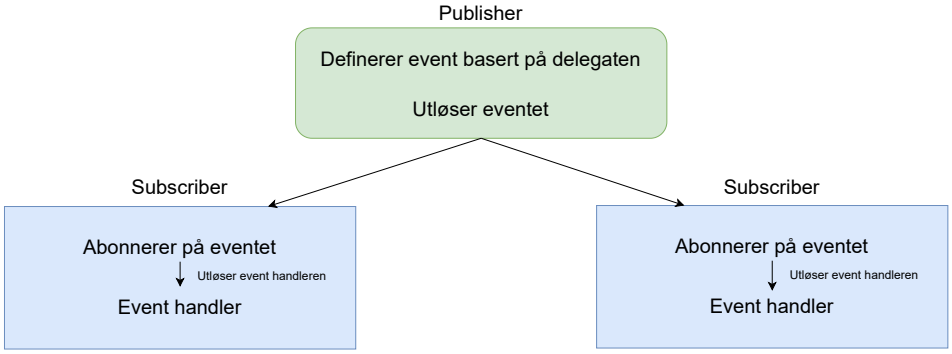
av C#-koden. Oversettelsen av kompilert C#-kode til CIL utfører C#-kompilatoren. Når CIL-filen skal kjøres tar .NET runtime over og oversetter til maskinkode ved bruk av CLR (Common language runtime). Denne oversettelsen skjer når programmet kjører. [5, s. 8]

2.1.3 NuGet-pakker

NuGet er en pakkehåndterer for .NET-miljøet. En pakke er en samling av kode laget av andre utviklere. Denne pakken kan deretter deles og brukes av andre utviklere. Dette gjør det mulig å gjenbruke andre utvikleres kode uten å måtte komme opp med grunnleggende eller avanserte prinsipper selv. NuGet håndterer hvordan pakker for .NET lages, deles og brukes. [15]

2.1.4 Events

Events er blitt hyppig brukt i utviklingen av applikasjonen og gjør det mulig for ulike komponenter i applikasjonen å snakke sammen uten at de har noe med hverandre å gjøre. Et event er en melding som utløses av et objekt og blir sendt når en handling inntreffer i applikasjonen. Deretter kan et annet objekt lytte på et event og utføre en spesiell operasjon når dette eventet har blitt utløst. Dette fører til en god samhandling mellom ulike klasser i applikasjonen, uten at de forskjellige klassene trenger å vite noe om hverandre. Figur 2.2 viser hvordan en *Publisher* definerer et event og hvordan en *Subscriber* abonnerer på eventet.



Figur 2.2: Events i C#.

2.1 Programmeringsspråk og rammeverk for utvikling av applikasjonen

Under viser et kodeeksempel på hvordan en *Publisher*- og *Subscriber*-klasse defineres i C#. Kode 2.1 viser *Publisher*-klassen hvor eventet blir definert med *EventHandler*-delegatet. Deretter blir en metode for å utløse eventet definert.

```
1  public class Publisher
2  {
3      // Definerer eventet basert pa EventHandler-delegat
4      public event EventHandler EnkeltEvent;
5
6      // Metode for a utlose eventet
7      public void UtlosEvent()
8      {
9          Console.WriteLine("Utloser event!");
10         EnkeltEvent?.Invoke(this, EventArgs.Empty);
11     }
12 }
```

Kode 2.1: Publisher-klassen.

Kode 2.2 viser *Subscriber*-klassen. Den har en enkel metode for å håndtere eventet.

```
1  public class Subscriber
2  {
3      // Metode for a handtere eventet
4      public void PaEnkeltEvent()
5      {
6          Console.WriteLine("Eventet blir utlost!");
7      }
8  }
```

Kode 2.2: Subscriber-klassen.

Kode 2.3 viser hovedprogrammet hvor eventene blir definert. Deretter abonneres det på eventet og til slutt blir eventet utløst.

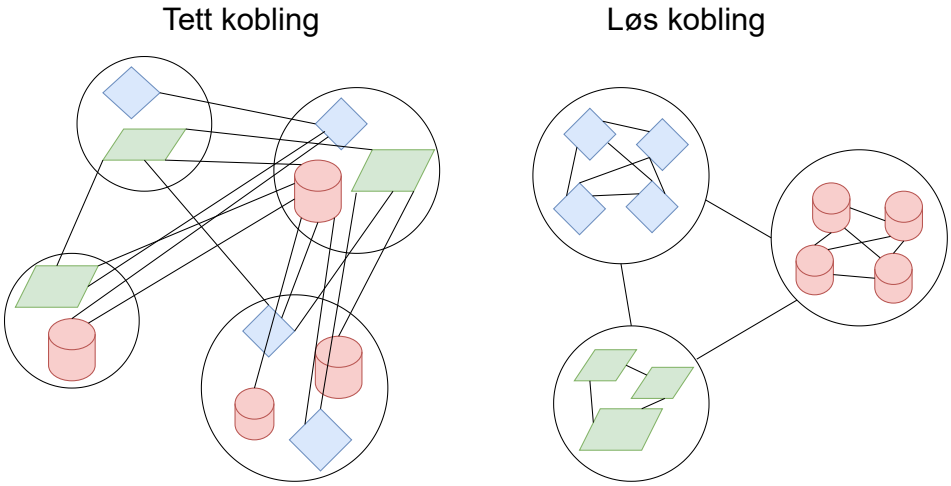
```
1  Publisher publisher = new Publisher();
2  Subscriber subscriber = new Subscriber();
3
4  // Abonnerer pa hendelsen
5  publisher.EnkeltEvent += subscriber.PaEnkeltEvent;
6
```

```
7 // Utloser eventet
8 publisher.UtlosEvent();
```

Kode 2.3: Hovedprogram og definering av events.

Fordel med events

Events bidrar til å løsne koblingen mellom komponenter. Det vil si at komponentene i koden ikke trenger å vite noe om hverandres implementasjon. Når en klasse definerer et event sier den bare hva som skal skje. Deretter kan en annen klasse abonnere på dette eventet og utføre sin egen logikk når hendelsen oppstår. Figur 2.3 illustrerer forskjellen på tett koblet og løst koblet kode.

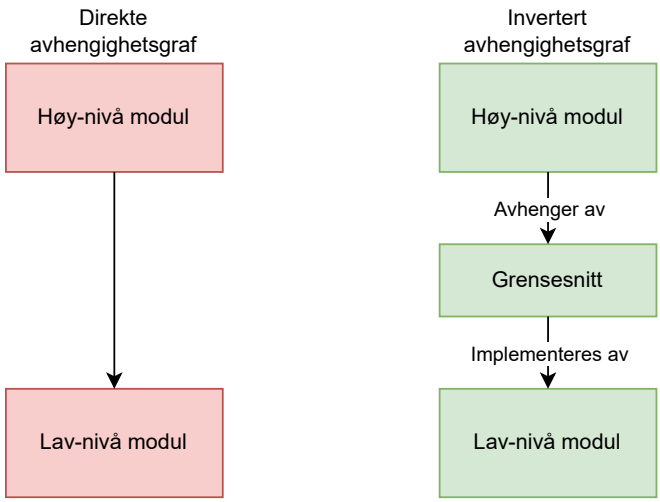


Figur 2.3: Forskjellen på tett og løst koblet kode.

2.1.5 Dependency injection

Dependency injection er et designmønster i programvaren som fører til dependency inversion. Tradisjonelt i programvareutvikling er det vanlig at høy-nivå moduler som ofte håndterer kompleks logikk, er direkte avhengig av lav-nivå moduler som ofte håndterer detaljerte operasjoner. Dette fø-

rer til tett koblet kode, noe som er vanskelig å teste og vedlikeholde. Med dependency inversion-prinsippet blir denne måten å utvikle på invertert. Dette fører til løst koblet kode. I motsetning til at høy-nivå moduler er direkte koblet med lav-nivå moduler, avhenger begge av abstraksjoner i form av grensesnitt eller abstrakte klasser. [13] Figur 2.4 viser forskjellen på tradisjonell programvareutvikling hvor høy-nivå moduler er direkte koblet til lav-nivå moduler i form av en direkte avhengighetsgraf. Dette fører til kode som er vanskelig å teste og tett koblet kode. Figuren viser også det ønskelige utfallet, hvor høy-nivå modulen avhenger av et grensesnitt som blir implementert av lav-nivå modulen, i form av en invertert avhengighetsgraf.



Figur 2.4: Grafisk presentasjon av dependency inversion.

Koden under viser hvordan dependency injection kan bli brukt i C#. Kode 2.4 viser opprettelsen av et grensesnitt *IMotor*, som høy-nivå modulen skal avhenge av. Grensesnittet viser kun hvilke metoder som skal finnes i høy-nivå modulen. Det er først når høy-nivå modulen *Motor* blir opprettet, at funksjonaliteten i metoden blir implementert. Motorklassen implementerer *IMotor*-grensesnittet ved bruk av kolon-syntaksen.

```
1 public interface IMotor
2 {
3     void StartMotor();
4 }
5
```

```
6  public Motor : IMotor
7  {
8      public void StartMotor()
9      {
10         Console.WriteLine("Motoren starter ...");
11     }
12 }
```

Kode 2.4: Opprettelsen av motorklassen - dependency injection.

Kode 2.5 viser hvordan bilklassen bruker *IMotor*-grensesnittet og hvor dependency injection kommer i bruk. Her blir en bilklasse implementert og bruker dermed motorklassen. For å unngå direkte avhengighet blir ikke en egen instans av motorklassen definert i bilklassen. Derimot blir kun grensesnittet definert og deretter blir motorklassen injisert gjennom konstruktøren.

```
1  public class Bil
2  {
3      private IMotor _motor;
4
5      // Konstruktorinjeksjon
6      public Bil(IMotor motor)
7      {
8          _motor = motor;
9      }
10
11     public void StartBil()
12     {
13         _motor.StartMotor();
14     }
15 }
```

Kode 2.5: Opprettelsen av motorklassen - dependency injection.

Kode 2.6 viser hvordan motorklassen blir injisert utenfor bilklassen. En ny instans av motorklassen blir definert og deretter sendt inn i konstruktøren av bilklassen.

```
1          IMotor motor = new Motor();
2          Bil bil = new Bil(motor);
3          bil.StartBil();
```

Kode 2.6: Bruk av dependency injection i programmet.

Levetid til tjenester

En fundamental del av programvarekonseptet dependency injection er at tjenester kan ha ulik levetid. De tre levetidene er:

- **Transient**: Vil bli opprettet hver gang den blir forespurt
- **Scoped**: Oppretter en ny instans innenfor hver scopet. For eksempel i en webapplikasjon, vil scopet være hver enkelt forespørsel til en server
- **Singleton**: Blir opprettet første gang den blir forespurt eller av utvikleren. Kun en instans under levetiden til applikasjonen

Valget mellom de ulike levetidene til tjenester vil avhenge av applikasjonens behov. [16]

Dependency injection i .NET

I .NET finnes det rammeverk for dependency injection som gjør bruken enklere. Kode 2.7 viser hvordan dependency injection kan bli konfigurert med .NET. *AddHostedService*-koden legger til bilklassen som en hostet tjeneste. Hostede tjenester gjør det mulig å strukturere bakgrunnstjenester i applikasjonen. [20] *AddSingleton*-koden registrerer *IMotor*-grensesnittet og motorklassen som en singleton-tjeneste. Deretter blir hosten konstruert med alle konfigurasjoner og kjørt. Dermed kan motorklassen bli brukt til dependency injection andre steder i programmet ved hjelp av automatisk dependency injection fra .NET. [16]

```
1 class Program
2 HostApplicationBuilder builder = Host....
      CreateApplicationBuilder(args);
3
4 builder.Services.AddHostedService<Bil>();
5 builder.Services.AddSingleton<IMotor, Motor>();
6
7 using IHost host = builder.Build();
8
9 host.Run();
```

Kode 2.7: Konfigurasjon av dependency injection i .NET.

2.2 XAML - Markeringsspråk for utvikling av brukergrensesnitt

XAML er et deklarativt markeringsspråk som i kombinasjon med .NET gjør det enklere å utvikle et brukergrensesnitt. Et deklarativt språk fokuserer på hva som skal presenteres i brukergrensesnittet, i stedet for hvordan det skal gjøres. XAML er brukt i en rekke rammeverk, deriblant Windows Presentation Foundation (WPF), Universal Windows Platform (UWP) og WinUI (Windows UI Library). Ved bruk av XAML oppstår det en separasjon mellom brukergrensesnittet av applikasjonen og det funksjonelle. Med denne metoden oppnår man et tydelig skille mellom designere og utviklere. Designere kan fokusere på applikasjonens utseende og brukeropplevelsen med XAML-kode, mens utviklere kan fokusere på applikasjonens logikk i bakgrunnskodefiler (code behind files). Dette bidrar til en enklere arbeidsprosess og økt skalerbarhet av applikasjonen, ettersom endringer i brukergrensesnittet kan gjøres uten å påvirke applikasjonens logikk og endringer i applikasjonens logikk kan gjøres uten å påvirke brukergrensesnittet. [21]

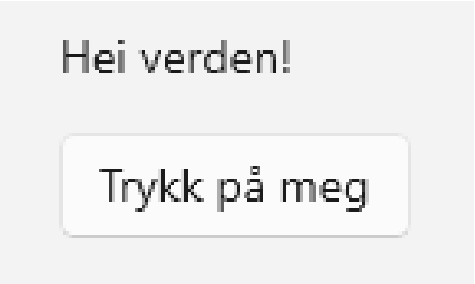
Kode 2.8 viser et eksempel på en XAML-fil brukt i en WPF-applikasjon. Filen definerer et *Window*-element. Dette elementet representerer hovedvinduet i applikasjonen. Deretter benyttes *Grid*-elementet for å organisere inneholdet. Innenfor *Grid*-elementet finnes det en knapp og en tekstblokk. Tekstblokken inneholder tekst som vises i applikasjonen, mens knappen er et interaktivt element. Det vil si at brukeren kan utløse et event ved å trykke på knappen. Metoden som utløses når knappen blir trykket på defineres ved *Click*-nøkkelordet og deretter navnet på metoden.

```
1 <Window x:Class="MinApplikasjon.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/...
              xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx...
              /2006/xaml"
4         Title="MainWindow">
5         <Grid>
```

```
6             <Button x:Name="knapp" Content="Trykk pa meg...
                  " Click="knapp_trykk"/>
7             <TextBlock x:Name="tekst" Text="Hei verden!"...
                  />
8         </Grid>
9 </Window>
```

Kode 2.8: MainWindow.xaml - XAML-fil.

Figur 2.5 viser hvordan knappen fra kode 2.8 ser ut i applikasjonen.



Figur 2.5: Hvordan knappen ser ut i applikasjonen.

Kode 2.9 viser bakgrunnskodefilen til XAML-filen. Denne filen styrer oppførselen til brukergrensesnittet. Koden initialiserer først brukergrensesnittet og definerer deretter metoden som knappen er linket til i XAML-filen. Metoden utfører logikken som skal skje når knappen blir trykket på. I dette eksempelet vil teksten på knappen endre seg.

```
1 using System.Windows;
2
3 namespace MinApplikasjon
4 {
5     public partial class MainWindow : Window
6     {
7         public MainWindow()
8         {
9             InitializeComponent();
10        }
11
12        private void knapp_trykk(object sender, ...
              RoutedEventArgs e)
13        {
```

```
14                knapp.Text = "Knapp ble trykket pa!";
15            }
16        }
17    }
```

Kode 2.9: MainWindow.xaml.cs - Bakgrunnskodefil.

Ved å kombinere XAML-filer og bakgrunnskodefiler på denne måten, muliggjør det en effektiv utvikling av responsive brukergrensesnitt.

2.3 UNO Platform - Multiplattform teknologi

UNO Platform er en åpen kildekode plattform, som muliggjør utviklingen av applikasjoner på flere plattformer med kun en kodebase. Denne teknologien åpner muligheten for at koden kun blir skrevet en gang og deretter vil programmet kunne kjøres på ulike operativsystemer og enheter. Spesifikt i denne oppgaven er målet at applikasjonen skal kunne kjøres som Windows-applikasjon og i nettleseren. [26]

2.3.1 UNO ved bygging av applikasjonen

UNO Platform-applikasjoner utvikles i C# og .NET-miljøet. En kombinasjon av C#- og XAML-kode, gjør det mulig å utvikle avanserte applikasjoner. Ved byggetid (run time) kan C#-koden kjøres direkte på ulike plattformer ved hjelp av .NET. XAML-filer blir derimot konvertert til C#-kode, noe som gir en jevn opplevelse for brukeren. [27]

2.3.2 Windows-applikasjon

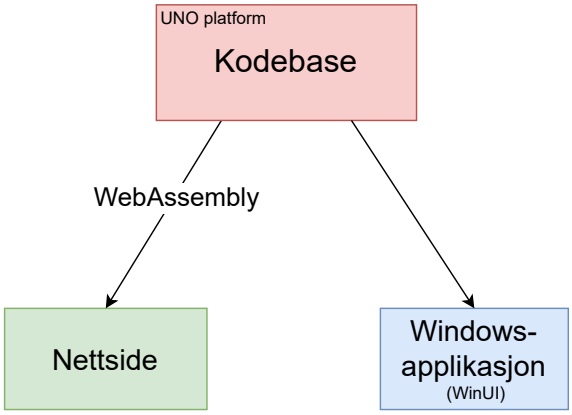
Siden UNO Platform bruker C# og .NET som sin kildekode, trenger ikke UNO å konvertere denne koden for å kunne kjøre applikasjonen på Windows. Dermed blir applikasjonen kompilert som en enkel plattformspesifikk WinUI-applikasjon. WinUI er et moderne og åpent kildekode-basert brukergrensesnittsbibliotek utviklet av Microsoft. Hensikten med WinUI

er å hjelpe utviklere med å bygge Windows-applikasjoner med et konsistent utseende som kan kjøres på ulike Windows-plattformer. Dette vil si at UNO-applikasjonen bruker Microsofts egne verktøy for å bygge og kjøre på Windows. [27] [19]

2.3.3 Nettside

UNO Platform gjør det mulig å kjøre applikasjonen i nettleseren uten å endre på kildekoden. C# samt XAML koden kompileres til WebAssembly. WebAssembly er en åpen webstandard som muliggjør at kode skrevet i C#, samt andre programmeringsspråk som C, C++ og Rust, kan kjøres på internett. Ved bruk av WebAssembly oppnås høy ytelse, sikkerhet og plattformuavhengighet i applikasjonen. Ved visning i nettleseren konverterer UNO XAML-elementer til webelementer. Dette innebærer å konverte eksempelvis paneler og kontroller fra XAML-kode til tilsvarende HTML-elementer. Ved bruk av UNO, som internt bruker WebAssembly, kan applikasjonen kjøres i nettleseren uten å endre på kildekoden, samtidig som høy ytelse opprettholdes. [27] [30]

Figur 2.6 viser en oppsummering av hvordan UNO kan brukes til å distribuere den samme kodebasen som både nettside og Windows-applikasjon.



Figur 2.6: UNO Platform sin rolle ved å distribuere den sammen kodebasen som nettside og Windows-applikasjon.

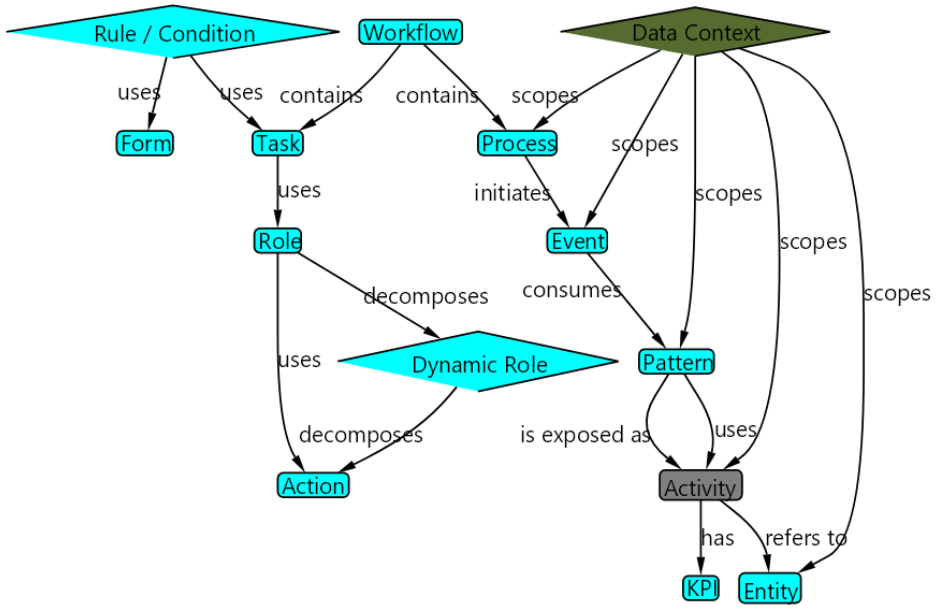
2.3.4 Hvorfor UNO Platform?

Det finnes mange muligheter ved valg av multiplattform-teknologi. Flutter og React Native er to av de mest kjente multiplattform-teknologiene. Sammenlignet med UNO er disse teknologiene mer populære og tatt mer i bruk. Derimot var var det ønskelig fra ABB at applikasjonen skulle utvikles ved bruk av UNO. Dette på bakgrunn av:

- **Erfaring på ABB**: ABB Bryne har kompetanse og erfaring ved bruk av UNO. Det var derfor ønskelig fra deres side at UNO ble brukt i utviklingen av applikasjonen
- **Bleeding edge-teknologi**: UNO er hele tiden i utvikling og kommer med kontinuerlige oppdateringer. Dette gjør UNO til en spennende og attraktiv teknologi å bruke, ettersom nye muligheter i teknologien alltid er å finne
- **Microsoft-økosystemet**: UNO har et godt samspill med C# og .NET. Fordelen med bruk av C# og .NET er at teknologien er godt integrert i Microsoft sitt økosystem, noe som passer ypperlig ettersom applikasjonen skal kunne kjøres på Windows og i nettleseren

2.4 Microsoft Automatic Graph Layout - Automatisk bygging av graf

Microsoft Automatic Graph Layout (MsAGL) er et .NET verktøy og bibliotek for å lage og samhandle med komplekse grafer. En automatisk graflayout kan enten være en programvare eller en algoritme for å konstruere en graf, uten at brukeren må flytte på noder eller koblinger. Målet er å visuelt fremstille en graf på en lesbar måte, ved å estetisk strukturere nodene og gi en visuell representasjon av relasjonene. Figur 2.7 viser et eksempel på en graf konstruert med MsAGL.



Figur 2.7: Eksempelgraf konstruert med MsAGL.

2.4.1 MsAGL moduler

MsAGL har fire ulike moduler for opprettelse og samhandling med grafer. Modulene er som følger:

- **Core Layout engine**: Kjernefunksjonen i MsAGL. Brukes for å bygge oppsettet av grafen
- **Drawing module**: Tilfører mer funksjonalitet til grafen. Eksempelvis endring av farge og linjestil. I tillegg gir den tilgang på en node-, edge- og graf-klasse, som senere kan brukes til rendering
- **WPF control**: Muliggjør visualisering av grafen i en applikasjon. Inkluderer også andre funksjonaliteter som zoom, dra i visningen, fokus på noder med mer. Designet for å brukes i WPF-applikasjoner
- **Windows Forms Viewer control**: Samme funksjonalitet som WPF control-modulen, bare at den er designet for Windows Form applikasjoner

Ved bruk av de innebygde modulene i MsAGL, er det mulig å automatisk bygge en graf i applikasjon. Deretter kan grafen visualiseres i applikasjonen. [22].

2.4.2 Sugiyama algoritmen

For å tegne grafen i applikasjonen brukes Sugiyama algoritmen innad i MsAGL. Sugiyama algoritmen er en avansert metode for å strukturere hierarkiske strukturer til grafer. [24] Det avanserte bruksområdet til Sugiyma blir ikke brukt i denne oppgaven. Derimot ble algoritmen brukt for å fremstille grafen horisontalt, i stedet for vertikalt som MsAGL har som standard.

2.4.3 MsAGL kombinert med UNO Platform

Et problem med kombinasjonen av MsAGL og UNO Platform er at de to teknologiene ikke fungerer med hverandre. Hvis MsAGL brukes direkte i en UNO-applikasjon, vil ikke applikasjonen kunne kjøre. Problemet forekommer av at UNO bygger Windows-applikasjonen til å bli en WinUI-applikasjon som beskrevet i kapittel 2.3.2, mens MsAGL orginalt er designet for å kunne kjøres på Windows Forms- og WPF-applikasjoner. [22]

Løsningen på dette problemet ble å utvikle en NuGet-pakke som kombinerer de to teknologiene. Dette var allerede utviklet på ABB Bryne, slik at det kunne implementeres direkte i applikasjonen. Som beskrevet i kapittel 2.4.1 har MsAGL to moduler som visualiserer grafen i WPF- og Windows Forms-applikasjoner. NuGet-pakken lager en ny modul i MsAGL som gjør at den kan vises på WinUI-applikasjoner. Denne modulen bruker SkiaSharp for visualisering av grafen. SkiaSharp er et åpent kildekode 2D-grafikkbibliotek. [10]

2.5 XML - Markeringsspråk for bæring av informasjon

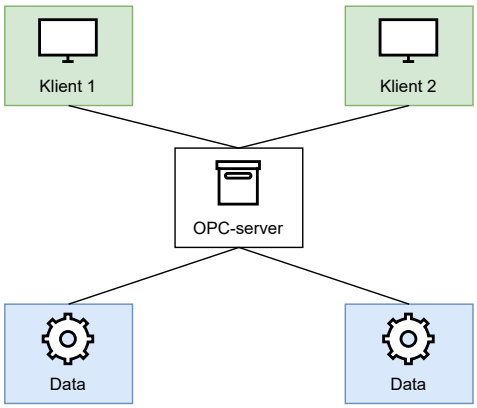
XML er et markeringsspråk som gjør det enkelt å bære informasjon i tillegg til å lagre metadata om informasjonen. Dette betyr at XML ikke gjør noe, det er kun er måte å lagre data på. En annen type programvare må derfor ta hånd av hvordan dataen skal vises, lagres eller endres. [29] XML blir brukt for å bære informasjonen om grafen som skal bygges i applikasjonen. Kode 2.10 viser et eksempel på en XML-fil, som inneholder informasjon og metadata om en graf.

```
<Graph>
  <Nodes>
    <Node id="1" label="Node 1" />
    <Node id="2" label="Node 2" />
  </Nodes>
  <Edges>
    <Edge source="1" target="2" />
  </Edges>
</Graph>
```

Kode 2.10: Eksempel på XML-kode.

2.6 OPC - Industristandard for kommunikasjon mellom ulike enheter

OPC (Open Platform Communications) er en industristandard for kommunikasjon mellom enheter fra ulike industrier. Standarden sikrer pålitelig veksling av data i industriell automatisering og andre industrier. OPC-standarden består av ulike spesifikasjoner som definerer grensesnittet mellom klienter og servere, samt servere og servere. Det er OPC foundation som er ansvarlig for utvikling og vedlikehold av denne standarden. Figur 2.8 viser hvordan klienter henter ut data gjennom en OPC-server. Datakilden kan være ulike enheter, som for eksempel en lakkeringsrobot.



Figur 2.8: Kommunikasjon ved bruk av OPC.

2.6.1 OPC Classic og OPC UA

OPC var opprinnelig begrenset til kun Windows-plattform applikasjoner. Denne spesifikasjonen er i dag kjent som OPC Classic. OPC Classic-spesifikasjonene er basert på Windows-teknologi ved bruk av COM/DCOM for å distribuere data mellom programvare komponenter. OPC Classic har ulike spesifikasjoner, dette innebærer:

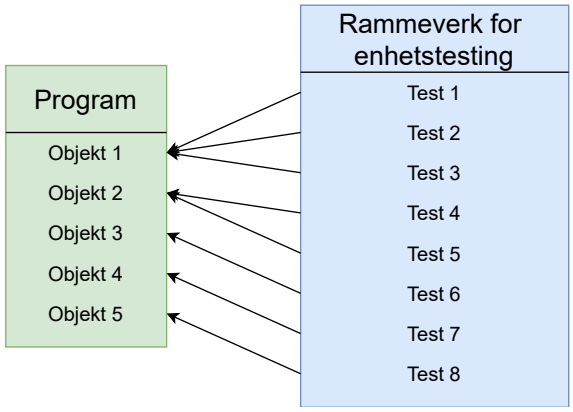
- **OPC Data Access**: Spesifikasjon for utveksling av data som verdier, tid og kvalitetsinformasjon
- **OPC Alarms and Events**: Spesifikasjon for utveksling av alarm og event type informasjon
- **OPC Historical Data Access**: Spesifikasjon for forespørselsmetoder og analyser av historiske- og tidsbestemte data

Ettersom teknologien utviklet seg, vokste også behovet for andre typer spesifikasjoner av OPC. Dermed utviklet OPC i 2008 en ny type teknologi kalt OPC UA (Unified Architecture). Denne teknologien videreførte alle mulighetene til OPC Classic, men det var ikke lenger kun tilgjengelig på Windows-applikasjoner. [7] Med OPC UA er OPC-teknologien tilgjengelig på flere plattformer som Windows, Apple OSX, Android og Linux. I tillegg

er OPC UA sikrere, ved at teknologien bruker kryptering, autentisering, og revisjon. [8]

2.7 Enhetstester - Testing av oppførsel i koden

Enhetstester er enkelte tester som har som oppgave å teste en spesifikk oppførsel i koden. Enhetstester skal i prinsippet kun teste en spesifikk oppøførsel og dermed svare på om denne oppførselen var suksessfull eller ikke. Det finnes mange rammeverk som hjelper til med utviklingen av enhetstester. Rammeverk for enhetstesting er programvare som skal hjelpe til å skrive og kjøre testene, samt gi et resultat på statusen av enhetstestene.[11, s. 1-2] Figur 2.9 viser sammenhengen mellom rammeverk for enhetstester og programmet. En test er linket med et objekt, eller med en del av et objekt. Det er også mulig å ha flere enhetstester på samme objekt.



Figur 2.9: Sammenhengen mellom rammeverk for enhetstester og programmet.

2.7.1 NUnit

I applikasjonen blir NUnit-rammverket brukt for enhetstesting. NUnit er et rammeverk for .NET-miljøet. Det betyr at rammeverket er skrevet i C# og støtter ethvert program som er skrevet i et .NET-språk. NUnit benytter seg av attributter i C# for å legge til metadata. Attributter er enkelte nøkkelord

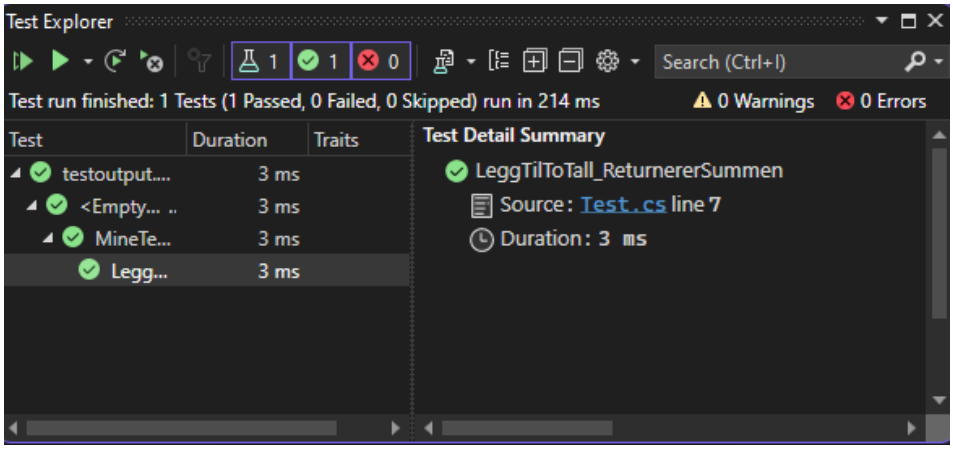
som blir plassert før et objekt eller en metode. NUnit sine viktigste attributter er *TestFixture* og *Test*. Attributten *TestFixture* blir plassert foran en klasse som inneholder flere tester, mens *Test* blir plassert foran en spesifikk testmetode.[11, s. 80-81]

Kode 2.11 viser et eksempel på hvordan en NUnit enhetstest kan defineres. Attributten *TestFixture* plasseres over definisjonen av klassen *MineTester*. Deretter plasseres attributen *Test* over den spesifikken testen. Denne testen summerer to tall og sjekker om svaret er lik en forventet sum. NUnit sitt nøkkelord *That* blir deretter brukt for å sjekke summen og om den forventede summen er det samme eller ikke.

```
1  using NUnit.Framework;
2
3  [TestFixture]
4  public class MineTester
5  {
6      [Test]
7      public void LeggTilToTall_ReturnerSummen
8      {
9          int tall = 1;
10         int tall2 = 2;
11         int forventet_sum = 3;
12
13         int resultat = tall + tall2;
14
15         Assert.That(forventet_sum == resultat);
16     }
17 }
```

Kode 2.11: NUnit enhetstest.

Figur 2.10 viser resultatet av enhetstesten. Resultatet viser at testen passerte og at koden fungerer.



Figur 2.10: Resultatet av enhetstesten.

2.8 BenchmarkDotNet

BenchmarkDotNet er et bibliotek som forenkler prosessen av å teste ytelsen av en metode, funksjon eller program. Biblioteket kan inkluderes i et program ved å innstallere det som en NuGet-pakke. Biblioteket er utviklet for å være brukervennlig og enkelt å komme i gang med, i tillegg til å gi nøyaktige resultater. BenchmarkDotNet har fire viktige egenskaper:

- **Enkelhet**: Biblioteket er utviklet til å være enkelt å bruke og gjør det lett å kjøre komplekse ytelsestester ved hjelp av enkle APIer
- **Automasjon**: Biblioteket automatiserer mange tidkrevende og repetetive oppgaver i henhold til ytelsestesting. Dette inkluderer pilotseksprimenter, oppvarmingsiterasjoner og hoveditерasjoner
- **Pålitelighet**: Biblioteket beskytter mot vanlige feil når det kommer til ytelsestesting. Dette gjør testene mer nøyaktige og pålitelige. I tillegg har også biblioteket mulighet for å gi advarseler hvis det er feil i miljøet som kan påvirke testene
- **Brukervennlighet**: Biblioteket er utviklet for å være brukervennlig og enkelt å komme i gang med. I tillegg presenteres resultatene på en

forståelig måte og muligheten til å eksportere resultatene til videre bruk

Ut i fra disse egenskapene er BenchmarkDotNet-biblioteket et pålitelig og brukervennlig bibliotek når det kommer til ytelsestester. [4]

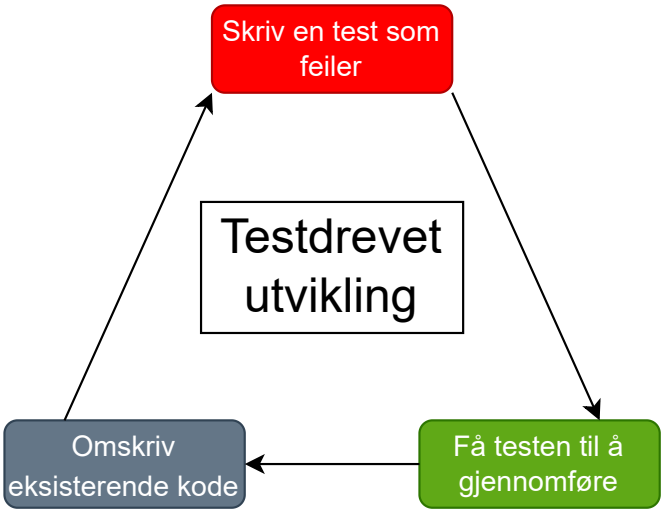
Kapittel 3

Konstruksjon

Kapittelet tar for seg hvordan applikasjonen er konstruert. Dette inkluderer metoder for utvikling samt struktur, funskjonalitet og fundamental kode bak applikasjonen.

3.1 Testdrevet utvikling

Testdrevet utvikling er en metode for utvikling av programmer. Testdrevet utvikling innebærer å skrive enhetstester for spesifikke metoder, før selve metoden har blitt skrevet. Dette fører til at algoritmen eller oppførselen til metoden kan planlegges på forhånd. Etter at enhetstestene er skrevet kan metoden bli laget slik at den akkurat passerer enhetstesten. Deretter kan testene og metoden videreutvikles, uten fare for at tidligere oppførsel endrer seg. Figur 3.1 viser en grafisk presentasjon av prosessen rundt testdrevet utvikling. Det starter med å skrive en test som feiler. Deretter bygges metoden som testen er skrevet for, slik at testen gjennomfører. Til slutt blir eksisterende kode omskrevet og metoden blir videreutviklet. Fordelen med testdrevet utvikling er at koden får økt kvalitet, det reduserer fremtidige feil og det blir lettere å endre på koden uten å ødelegge annen funksjonalitet.



Figur 3.1: Grafisk presentasjon av testdrevet utvikling.

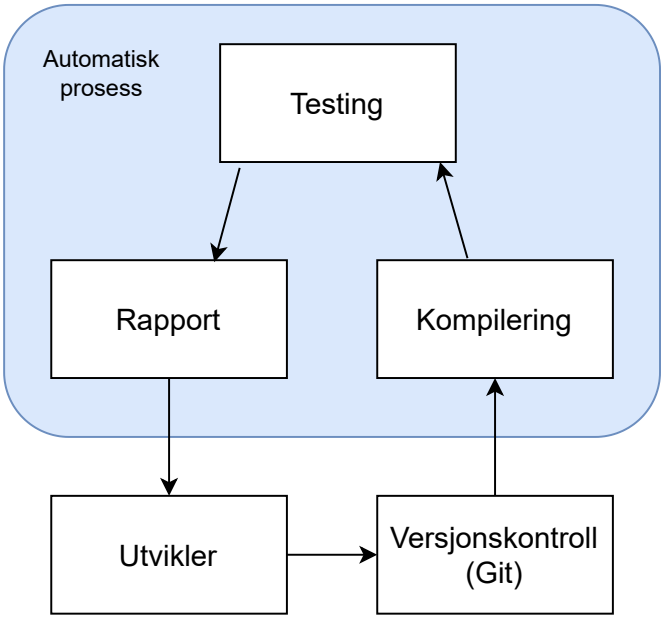
3.1.1 Bruk av testdrevet utvikling

Testdrevet utvkling ble tildels brukt i utviklingen av applikasjonen. Til viktige komponenter og deler av programmet ble det skrevet tester før de ble utviklet. Dette ble gjort siden det var vanskelig å vite nøyaktig hvor utviklingen skulle starte. Dermed gjorde testdrevet utvikling denne prosessen enklere ved at ønsket oppførsel ble først testet og deretter ble logikken bak skrevet. På enkle klasser og konfigurasjonskode ble det ikke skrevet tester i forveien, ettersom dette ikke var nødvendig. Det finnes andre utviklingsmetoder, men i omfanget av denne oppgaven ble kun testdrevet utvikling benyttet.

3.2 Kontinuerlig integrasjon

Kontinuerlig integrasjon er en programmeringsmetode som innebærer å ofte sende inn kildekoden til et delt repository. Deretter blir koden som blir sendt inn, automatisk kjørt og testet for eventuelle feil. Fordelen med dette er at utvikleren ikke trenger å bruke mye tid på feilsøking i koden og kan heller fokusere på å skrive kode. I tillegg blir koden automatisk testet ved

hver innsending. [9] Figur 3.2 viser en grafisk presentasjon av prossesen kontinuerlig intergasjon.



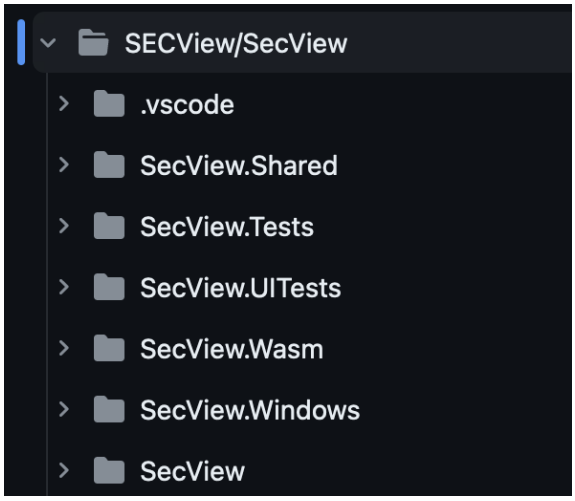
Figur 3.2: Grafisk presentasjon av kontinuerlig integrasjon.

3.2.1 GitHub actions

I utviklingen av applikasjonen ble GitHub actions brukt for kontinuerlig integrasjon. Dette innebærer at ved hver innsending av kildekode til det delte repositoriet på GitHub, vil GitHub automatisk kjøre koden og alle testene som er skrevet. GitHub kan deretter bli brukt til å sjekke status på programmet og om testene feilet eller ikke.

3.3 Struktur i applikasjonen

Når prosjektet opprettes, deler UNO Platform automatisk applikasjonen inn i separate mapper. Figur 3.3 viser strukturen til prosjektet på GitHub.



Figur 3.3: Strukturen til prosjektet på GitHub.

Mappene inneholder følgende:

- **Shared**: Inneholder delte ressurser som brukes på alle plattformer
- **Tests**: Enhetstester for applikasjonen
- **UITests**: Tester som verifiserer at brukergrensesnittet oppfører seg likt på tvers av ulike plattformer
- **Wasm**: Spesifikk kode og ressurser for WebAssembly
- **Windows**: Spesifikk kode og ressurser for Windows
- **Hovedmappe (SecView)**: Kodebasen. Her applikasjonen utvikles fra

Hovedmappen er deretter delt inn i ulike undermapper for å opprettholde god struktur i applikasjonen. Hovedmappen er innledningsvis delt opp i to mapper:

- **Core**: Hovedfunksjonalitet i applikasjonen
- **Mocking**: Funksjonalitet for å simulere data

Mappen *Core* er deretter delt opp i følgende mapper:

- **Events**: Klasser som definerer events
- **Exceptions**: Klasser som definerer egne exceptions
- **FileUpload**: Klasser relatert til filopplastning
- **GraphCreation**: Klasser relatert til bygging av grafen
- **NodeInfo**: Klasser som inneholder informasjon om nodene

3.4 Konfigurasjonsfil for grafen

For å løse målet om en dynamisk opprettelse av den grafiske visningen av sikkerhetstiltakene, falt vaglet på å la brukeren ha muligheten til å laste opp en konfigurasjonsfil. Konfigurasjonsfilen skal representere alle sikkerhetstiltakene angitt som noder i en graf. Filen skal også inneholde alle relasjoner mellom ulike sikkerhetstiltak, samt metadata om hvert sikkerhetstiltak.

For at applikasjonen skal klare å bygge grafen gjennom MsAGL må en viss struktur opprettholdes i konfigurasjonsfilen, representert som en XML-fil. XML-filen må inneholde fem nøkkelord, *graph*, *nodes*, *node*, *edges* og *edge*. *Nodes* består av flere *node* og *edges* består av flere *edge*. Kode 3.1 viser et eksempel på en XML-fil som kan lastes opp i applikasjonen.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <graph>
3      <nodes>
4          <!-- Line 1-->
5          <node id="1" label="Panel" firstnode="true" ...
               shape="box">
6          <node id="2" label="envy stop relay" lastnode="...
               true" shape="ellipse">
7
8          <!-- Line 2-->
9          <node id="3" label="motor off" firstnode="true" ...
               shape="box">
10         <node id="4" label="motor on status" lastnode="...
               true" shape="ellipse">
```

```
11      </nodes>
12
13      <edges>
14          <!-- Edges for line 1-->
15          <edge source="1" target="2"></edge>
16
17          <!-- Edges for line 2-->
18          <edge source="3" target="4"></edge>
19      </edges>
20  </graph>
```

Kode 3.1: Eksempel på XML-fil som kan bli lastet opp i applikasjonen.

Hver *node* må inneholde en unik *id* og kan ha et navn gitt ved nøkkelordet *label*. I tillegg må første node i en kjede ha verdien *firstnode="true"*, samt siste node i en kjede må ha verdien *lastnode="true"*. Dette gjør at det er mulig å ha flere kjeder som er uavhengige av hverandre. Hver *edge* består av to nøkkelord, *source* og *target*. Nøkkelordet *source* gir utgangspunktet for koblingen mellom to noder, mens nøkkelordet *target* gir destinasjonsnoden.

3.4.1 Dynamisk opprettelse av konfigurasjonsfil ved hjelp av Python

Et Python-skript ble laget for å unngå å skrive konfigurasjonsfilen manuelt. Dette gjorde det mulig å teste applikasjonen med større konfigurasjonsfiler. Skriptet ble ikke brukt direkte i applikasjonen, men var et viktig verktøy når applikasjonen skulle bli testet. Ved opprettelsen av konfigurasjonsfilen var følgende egenskaper ønsket fra Python-skriptet:

- Antall kjeder (uavhengige grafer) skal kunne velges
- Antall noder i hver kjede skal være et tilfeldig tall mellom to valgte verdier
- Alle noder må ha en *id*
- Hver node skal ha en tilfeldig *label*-verdi, tatt fra en liste med verdier som kan settes
- Siste node i en kjede må ha attributten *lastnode* satt til *true*

- Første node i en kjede må ha attributten *firstnode* satt til *true*
- Siste node i en kjede skal ha symbolet *ellipse*
- Alle andre noder skal ha et tilfeldig symbol, tatt fra en liste med symboler som kan settes

Vedlegg A viser hele Python-skriptet for dynamisk opprettelse av konfigurasjonsfiler.

3.5 Simulering av data

Etter en konfigurasjonsfil er blitt lastet opp i applikasjonen, er det også mulig å laste opp en CSV-fil. CSV-filen skal simulere direkte innsendte data fra en robot. Filen blir lastet opp av brukeren selv og fungerer primært som en funksjonalitet for å teste applikasjonen. Simulering av data gjør det mulig å teste funksjonaliteten i programmet, uten å koble seg opp til en robot. For at applikasjonen skal kunne ta inn simulerte signaler fra CSV-filen, må den bestå av følgende tre kolonner med data:

- **NodeId**: *Id* til en gitt node
- **DateSignalChange**: Dato med klokkeslett som angir når signalet skal inntreffe
- **Signal**: Hvilket signal som skal sendes. Enten 0 eller 1, hvor 0 er en feil på noden og 1 er ingen feil

Tabell 3.1 viser et eksempel på strukturen i en CSV-fil, som sender simulerte signaler til applikasjonen på et gitt klokkeslett og dato.

Tabell 3.1: Struktur i CSV-filen som simulerer innsendt data til applikasjonen.

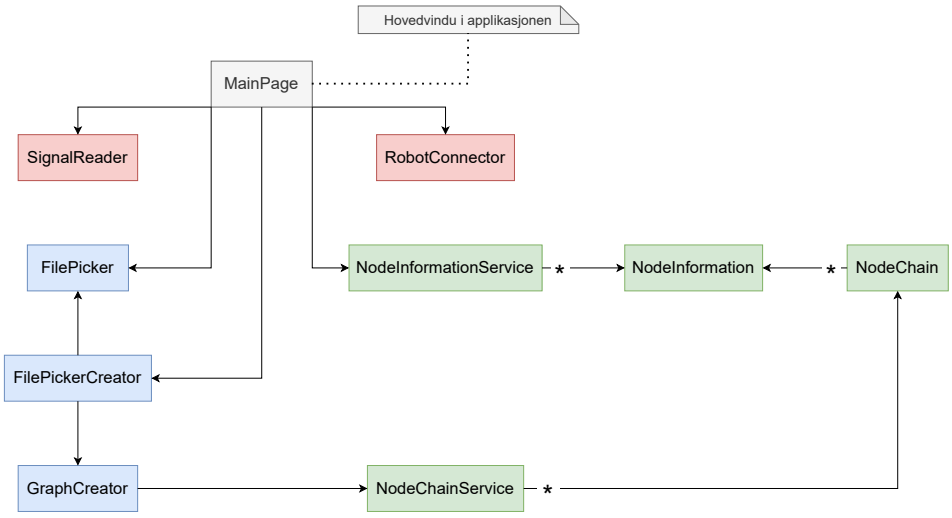
NodeId	DateSignalChange	Signal
1	01.01.2025 12:12:00	0
2	01.01.2025 12:12:30	1
3	01.01.2025 12:13:00	0
4	01.01.2025 12:13:05	0
5	01.01.2025 12:13:05	1

3.6 Komponenter - Sentrale klasser og grensesnitt i applikasjonen

Applikasjonen består av mange store og små klasser, objekter og grensesnitt som har ulike felter, metoder og avhengigheter av hverandre. De viktigste komponentene i applikasjonen blir herved listet og forklart detaljert.

3.6.1 Applikasjonens arkitektur

Figur 3.4 viser et UML klassediagram for hele applikasjonens arkitektur.



Figur 3.4: Klassediagram for applikasjonens arkitektur.

MainPage er hovedsiden i applikasjonen og samhandler med følgende underliggende klasser:

- **SignalReader**: Leser data fra CSV-filer
- **RobotConnector**: Kobler seg opp mot roboten og får innsendt live data
- **FilePicker**: Lager visningen for valg av filer
- **FilePickerCreator**: Lager knappen som åpner visningen for valg av filer
- **NodeInformationService**: Tilbyr ulike tjenester for samhandling med *NodeInformation*-klassen

FilePickerCreator-klassen samhandler med følgende klasser:

- **FilePicker**
- **GraphCreator**: Lager grafen etter en konfigurasjonsfil har blitt lastet opp

GraphCreator-klassen samhandler med følgende klasser:

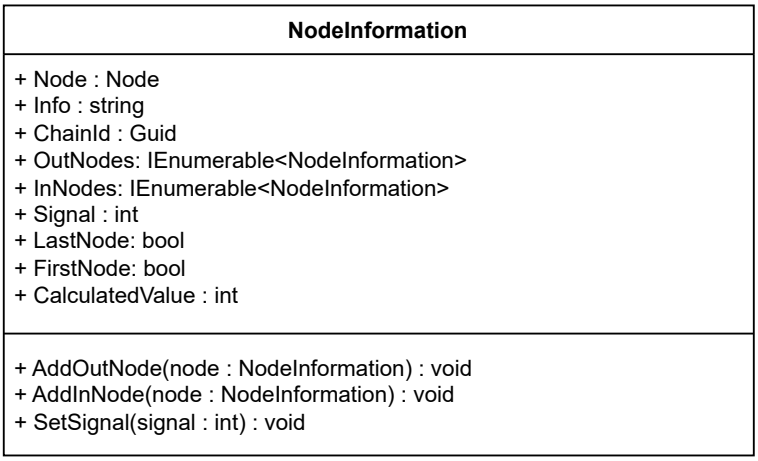
- **NodeChainService**: Tilbyr ulike tjenester for samhandling med *NodeChain*-klassen

NodeChainService-klassen samhandler med *NodeChain*-klassen. *NodeChain*-klassen består av flere *NodeInformation*-objekter.

3.6.2 NodeInformation - Informasjon om node

Klassen *NodeInformation* er den mest essensielle og sentrale klassen i programmet. Klassen lagrer informasjon om hver node og brukes til å utløse

events og endre informasjon på en gitt node i grafen. Figur 3.5 viser et overordnet klassediagram av *NodeInformation*-klassen.



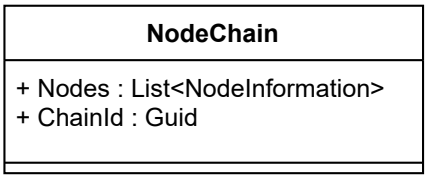
Figur 3.5: Klassediagram for *NodeInformation*-klassen med offentlige metoder og felter.

For å opprettholde sammenhengen mellom forskjellige noder holder hver *NodeInformation*-instans to lister med *NodeInformation*-klasser. Den ene listen inneholder koblingen mellom nåværende node og link til neste node(r). Den andre listen inneholder koblingen mellom nåværende node og link til tidligere node(r).

- **AddOutNode()**: Legger til ny node i listen, som inneholder kobling til neste node(r)
- **AddInNode()**: Legger til ny node i listen, som inneholder kobling til tidligere node(r)
- **SetSignal()**: Sjekker om innkommende signal er annerledes enn tidligere signal på en gitt node. Hvis det er annerledes settes korrekt farge i henhold til signalet på noden og et event utløses

3.6.3 NodeChain - Kjede med flere noder

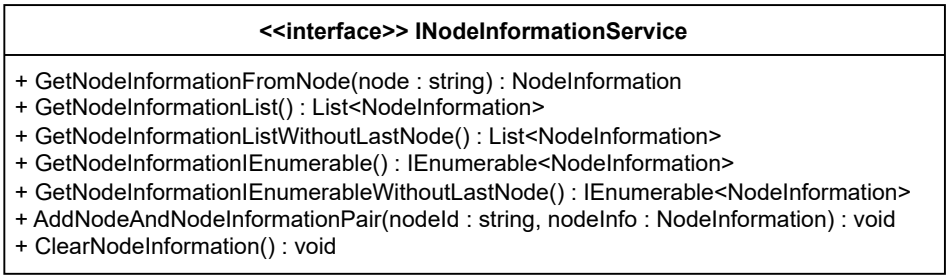
Klassen *NodeChain* er en egen klasse som består av en liste med *NodeInformation* objekter og en egen *id*. Klassen brukes til å definere kjeder i grafen, slik at det er mulig å ha flere kjeder innad i grafen. Figur 3.6 viser et klassediagram av *NodeChain*-klassen.



Figur 3.6: Klassediagram for *NodeChain*-klassen med offentlige felter.

3.6.4 NodeInformationService - Tjeneste for informasjon om en node

Klassen *NodeInformationService* implementerer *INodeInformationService*-grensesnittet og er en tjeneste som har ulike metoder for å hente informasjon fra *NodeInformation*-klassen. Figur 3.7 viser et klassediagram av *INodeInformationService*-grensesnittet.



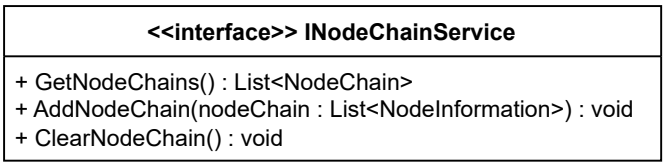
Figur 3.7: Klassediagram for *INodeInformationService*-grensesnittet med metoder.

Tjenesten *NodeInformationService* holder et privat *dictionary* for å holde oversikt over alle *NodeInformation*-objektene.

- **GetNodeInformationFromNode()**: Tar inn en *id* for en node, og returnerer et *NodeInformation*-objekt
- **GetNodeInformationList()**: Returnerer en liste med alle *NodeInformation*-objektene
- **GetNodeInformationListWithoutLastNode()**: Returnerer en liste med alle *NodeInformation*-objektene, uten å ta med den siste noden
- **GetNodeInformationIEnumerable()**: Returnerer en *IEnumerable* av alle *NodeInformation*-objektene
- **GetNodeInformationIEnumerableWithoutLastNode()**: Returnerer en *IEnumerable* av alle *NodeInformation*-objektene, uten å ta med den siste noden
- **AddNodeAndNodeInformationPair()**: Legger til et nytt par av *id* og *NodeInformation*-objekt i det private dictionary
- **ClearNodeInformation()**: Tømmer det private dictionary

3.6.5 NodeChainService - Tjeneste for en kjede med flere noder

Klassen *NodeChainService* er en tjeneste som implementerer *INodeChainService*-grensesnittet og står for behandling av kjeder i grafen. Figur 3.8 viser et klassediagram av *INodeChainService*-grensesnittet, *NodeChainService* implementerer.



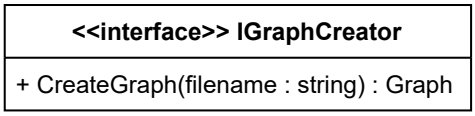
Figur 3.8: Klassediagram for *INodeChainService*-grensesnittet med metoder.

Klassen *NodeChainService* har en privat liste med uavhengige kjeder. Metodene *NodeChainService*-klassen implementerer fra grensesnittet har følgende funksjonalitet:

- **GetNodeChains()**: Henter ut alle kjedene fra listen over kjeder
- **AddNodeChain()**: Legger til en ny kjede i listen. Brukes når grafen bygges
- **ClearNodeChain()**: Tømmer listen, slik at det er mulig å laste opp en ny graf etter at en tidligere graf har blitt lastet opp

3.6.6 GraphCreator - Lesing av data fra XML-fil og opprettelse av grafobjekt

Klassen *GraphCreator* er en enkel klasse som implementerer *IGraphCreator*-grensesnittet. Metoden *CreateGraph* er den eneste metoden, utenom konstruktøren til *GraphCreator*-klassen. *CreateGraph*-metoden blir brukt til å lese inn data fra en XML-fil og returnerer deretter et grafobjekt. Figur 3.9 viser et klassediagram av *IGraphCreator*-grensesnittet.

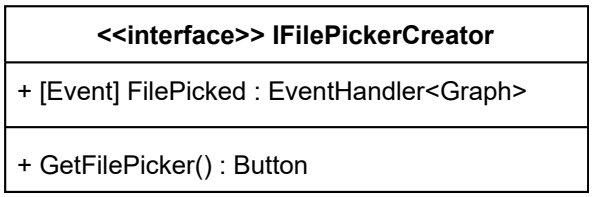


Figur 3.9: Klassediagram for *IGraphCreator*-grensesnittet med en metode.

- **CreateGraph()**: Metode som tar inn et filnavn som argument. Deretter brukes MsAGL til å lage et grafobjekt og instanser av klassene *NodeInformation* og *NodeChain*. Til slutt blir grafobjektet returnert som deretter kan vises i applikasjonen.

3.6.7 FilePickerCreator - Opplasting av XML-fil

Klassen *FilePickerCreator* implementerer *IFilePickerCreator*-grensesnittet og bruker den interne klassen *FilePicker*. Klassen oppretter en knapp som lar brukeren trykke på og laste opp en konfigurasjonsfil. Etter at en XML-fil har blitt lastet opp blir *GraphCreator*-klassen brukt videre for å håndtere filen. Figur 3.10 viser et klassediagram av *FilePickerCreator*-grensesnittet.

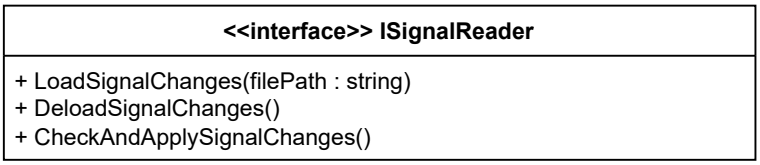


Figur 3.10: Klassediagram for *IFilePickerCreator*-grensesnitt med en metode og et event.

- **GetFilePicker()**: Metode for å lage knapp for opplastning av konfigurasjonsfil

3.6.8 SignalReader - Lesing av signaler

Klassen *SignalReader* implementerer *ISignalReader*-grensesnittet. Klassen leser, planlegger og utfører signalendringer. Dette skjer på spesifikke tidspunkt angitt i CSV-filen. Figur 3.11 viser et klassediagram av *ISignalReader*-grensesnittet.



Figur 3.11: Klassediagram for *ISignalReader*-grensesnittet med metoder.

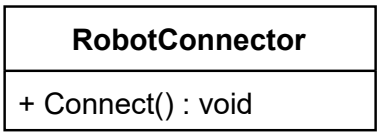
Klassen *SignalReader* holder en privat liste med *id* til en node, hvilket tidspunkt signalendring skal skje på og et signal. Denne listen brukes til å sende ut events til ulike noder om signalendring.

- **LoadSignalChanges()**: Initierer signalendringer fra fil. Leser inn en CSV-fil og setter opp tidspunkter og signaler
- **DeloadSignalChanges()**: Tømmer listen for alle planlagte signalendringer

- **CheckAndApplySignalChanges()**: Sjekker om signalendring har skjedd og sender deretter ut et event hvis en signalendring har skjedd

3.6.9 RobotConnector - Oppkobling til robot

Klassen *RobotConnector* er ansvarlig for å koble applikasjonen til OPC-serveren, som igjen kommuniserer med roboten. Klassen *RobotConnector* lytter på events fra OPC-serveren og sender ut events til nodene om en endring har skjedd. Siden klassen hele tiden må lytte etter nye events, er det essensielt at den samme instansen av klassen har levetid så lenge programmet kjører. Derfor er denne klassen statisk. Figur 3.12 viser et klassediagram av *RobotConnector*-klassen.



Figur 3.12: Klassediagram for *RobotConnector*-klassen med en metode.

Klassen *RobotConnector* bruker en pakke fra `https://www.opclabs.com/`. Nærmere bestemt pakken QuickOPC fra OPC Labs, for å samhandle med OPC-serveren i .NET.

- **Connect()**: Kobler seg opp mot roboten gjennom OPC-serveren, og sender deretter ut events ved signalendringer

3.7 Funksjonalitet og konstruksjon

Følgende delkapittel tar for seg utviklet funksjonalitet innad i applikasjonen og hvordan dette er blitt oppnådd. Figur 3.13 viser et overordnet bilde av applikasjonen, hvor komponentene i applikasjonen er markert med ulike farger.

Figur 3.13: Applikasjonen med funksjonalitet markert med farger.

- **Grønt område**: Hovedvinduet i applikasjonen. Her grafen vises etter at en konfigurasjonsfil har blitt lastet opp
- **Gult område**: Meny med ulike brukervennlige funksjonaliteter
- **Blått område**: Informasjonsboks hvor informasjon om noder dukker opp
- **Lilla område**: To radioknapper som gir muligheten til å velge mellom normal mode og panning mode
- **Rosa område**: Knapp for opplastning av konfigurasjonsfil
- **Rødt område**: Knapp for å skjule og vise menyen

3.7.1 Opplasting av konfigurasjonsfil

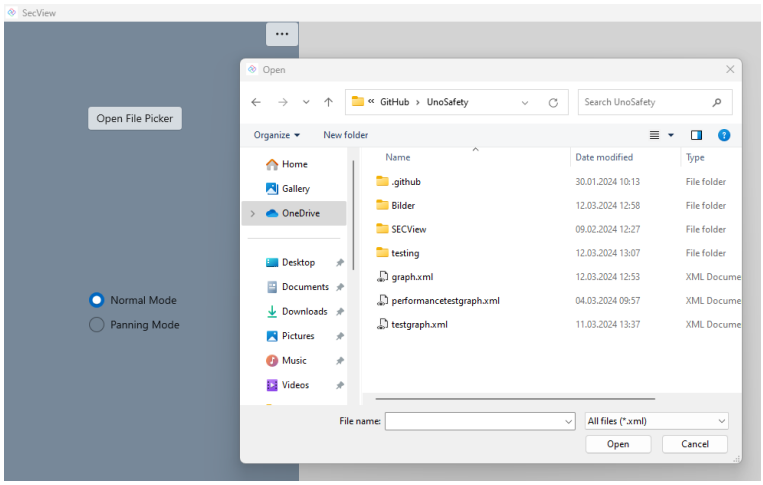
Ved oppstart av programmet vises ingen graf i applikasjonen. Det er mulig å vise en graf ved å laste opp en konfigurasjonsfil som gjøres ved å trykke

på *Open file picker*-knappen. Knappen blir laget i en privat metode i *FilePickerCreator*-klassen kalt *CreateUploadButton*. Her blir knappen laget og deretter gitt navnet *Open file picker*, som vist i kode 3.2.

```
1 Button uploadButton = new Button();
2 uploadButton.Content = "Open File Picker";
```

Kode 3.2: Opprettelse av *Open file picker*-knappen.

Etter at *Open file picker*-knappen blir trykket på, åpnes et nytt vindu hvor konfigurasjonsfilen for grafen kan bli valgt. Figur 3.14 viser vinduet som åpnes etter at *Open file picker*-knappen blir trykket på. I vinduet som åpnes vil kun gyldige filer, altså XML-filer, bli vist i mappene. Dette vil minke risikoen for at feil fil blir valgt.



Figur 3.14: Opplasting av konfigurasjonsfil.

For å oppnå denne oppførselen når knappen blir trykket på, blir den interne *FilePicker*-klassen brukt. Som vist i kode 3.3 blir først *OnOpenFilePickerClickedAsync*-metoden i *FilePicker*-klassen kalt. Først sjekker metoden om den valgte filen er *null*. Hvis den ikke er det, blir grafen laget. I tillegg sendes et event ut, for å signalisere at grafen har blitt laget.

```
1 await filePicker.OnOpenFilePickerClickedAsync(".xml",(...
      pickedFile) =>
```

```
2     {
3         if (pickedFile != null)
4         {
5             Graph graph = _graphCreator.CreateGraph(...
                  pickedFile.Path);
6             OnFilePicked(graph);
7         }
8     });
```

Kode 3.3: Logikk for opplastning av konfigurasjonsfil.

3.7.2 Visning av graf etter konfigurasjonsfil er lastet opp

Som angitt i kapittel 2.4 blir MsAGL-biblioteket brukt til å bygge grafen. I tillegg til å bygge grafen når en konfigurasjonsfil har blitt lastet opp, blir alle nødvendige klasser relatert til nodene opprettet automatisk av applikasjonen. Metoden *CreateGraph* i *GraphCreator*-klassen, henter ut all nødvendig informasjon fra konfigurasjonsfilen og oppretter instanser av *NodeInformation*-klassen og *NodeChain*-klassen.

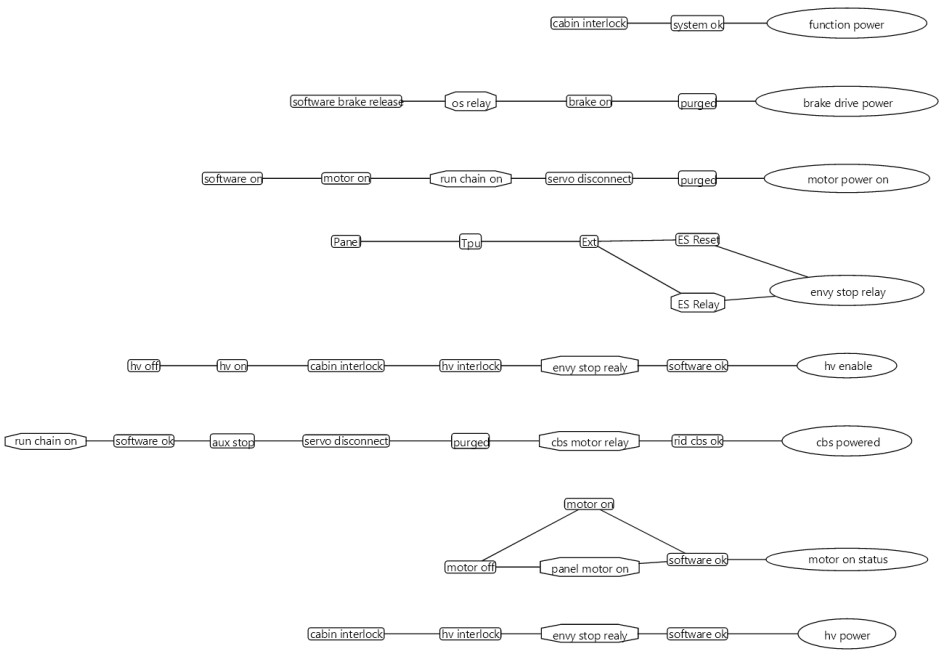
Kode 3.4 viser hovedkoden for bygging av grafen etter at en konfigurasjonsfil har blitt lastet opp. Merk at henting av data fra konfigurasjonsfilen og annen mindre viktig kode ikke blir vist i dette kodeutdraget. Først blir MsAGL-biblioteket brukt til å lage en instans av *Graph*- og *Node*-objektet. Instansen av *Node*-objektet blir brukt for å lage en instans av *NodeInformation*-klassen. Hvis det er siste node i en kjede, vil kjeden bli opprettet ved bruk av *NodeChainService*. Til slutt blir grafen laget ved bruk av MsAGL-biblioteket.

```
1 Graph graph = new Graph();
2 foreach (var nodeElement in xDocument.Descendants("node"...
      ))
3 {
4     string nodeId = nodeElement.Attribute("id").Value;
5     string nodeLabel = nodeElement.Attribute("label")....
          Value;
6     Node node = new Node(nodeId);
7
8     NodeInformation nodeInformation = new ...
          NodeInformation(node, nodeInfo, last, first, ...
          _signalReader);
```

```
9
10      if(last)
11      {
12          // for making the node chain
13          nodeChain.Add(nodeInformation);
14          _nodeChainService.AddNodeChain(nodeChain);
15          nodeChain = new List<NodeInformation>();
16      }
17      else
18      {
19          nodeChain.Add(nodeInformation);
20      }
21  }
22  graph.AddNode(node);
```

Kode 3.4: Opprettelse av graf når en gyldig fil har blitt opplastet.

Som vist i det grønne området angitt i figur 3.13, vises grafen her etter en konfigurasjonsfil har blitt lastet opp. Figur 3.15 viser et eksempel på en graf som har blitt bygget av en XML-fil.



Figur 3.15: Grafen i applikasjonen.

Standardoppsettet på en graf i MsAGL er å vise grafen i vertikal retning. Dette var ikke ønskelig, ettersom sikkerhetstilakene er horisontale i Safety Chain Viewer. For å oppnå dette ble Sugiyama algoritmen brukt for å rotere grafen, som angitt i kapittel 2.4.2. Kode 3.5 viser koden for å rotere grafen. Først blir algoritmen satt til Sugiyama. Deretter blir grafen rotert ved bruk av verdien $\frac{\pi}{2}$. Denne verdien er oppgitt i radianer, og vil derfor rotere grafen 90 grader. Dette vil føre til at grafen ligger horisontalt, i stedet for vertikalt.

```
1 var settings = new SugiyamaLayoutSettings();
2
3 settings.Transformation = PlaneTransformation.Rotation(...
      Math.PI / 2);
```

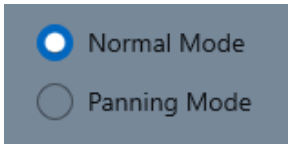
Kode 3.5: Endring av visning i graf fra vertikalt til horisontalt.

3.7.3 Brukerinteraksjon med visning i grafen

I menyen finnes det også to radioknapper som gjør det mulig å velge mellom to moduser. De to modusene er:

- **Normal mode**: Vanlig modus hvor det er mulig å klikke på nodene og zoome inn og ut
- **Panning mode**: Modus hvor det er mulig å zoome inn og ut samt dra i visningen. Det er ikke mulig å klikke på nodene i denne modusen

Figur 3.16 viser hvordan radioknappene ser ut i applikasjonen.



Figur 3.16: Radioknapper for de to modusene *normal-* og *panning mode*.

For å oppnå funksjonaliteten hvor man kan velge mellom de to modusene, finnes det to funksjoner i *MainPage.xaml.cs*-filen. Den første funksjonen er

TurnOnNormalMode som blir aktivert når *Normal mode*-knappen aktiveres. Den deaktiverer muligheten til å dra i visningen og aktiverer muligheten til å klikke på nodene. Zooming er standardfunksjonalitet i MsAGL og trenger derfor ikke aktiveres. Kode 3.6 viser *TurnOnNormalMode*-funksjonen.

```
1 _panningModeActivated = false;
2 var viewer = _gFrame.Content as GViewer;
3 viewer.LayoutEditingEnabled = true;
```

Kode 3.6: *TurnOnNormalMode*-funksjonen i *MainPage.xaml.cs*.

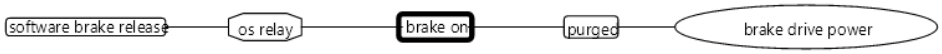
Den andre funksjonen er *TurnOnPanningMode* som blir aktivert når *Panning mode*-knappen aktiveres. Denne funksjonen aktiverer muligheten til å dra i visningen og deaktiverer muligheten til å klikke på nodene. Kode 3.7 viser *TurnOnPanningMode*-funksjonen.

```
1 _panningModeActivated = true;
2 var viewer = _gFrame.Content as GViewer;
3 viewer.LayoutEditingEnabled = false;
```

Kode 3.7: TurnOnPanningMode-funksjonen i *MainPage.xaml.cs*.

3.7.4 Informasjon om en gitt node

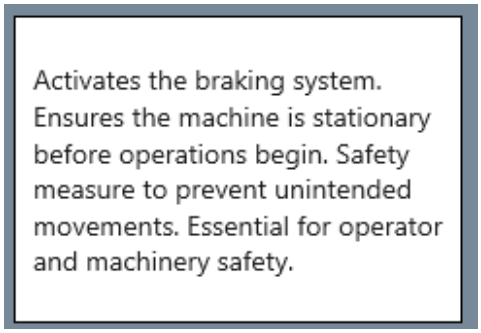
Når *normal mode* er valgt er det mulig å klikke på en node og få informasjon om den gitte noden. Når en node er blitt klikket på vil den bli markert ved å få en tykkere linje. Figur 3.17 viser en kjede i grafen hvor en node har blitt klikket på.



Figur 3.17: En gitt kjede hvor noden *brake on* har blitt klikket på og er dermed markert.

Informasjonen en node kan angis i konfigurasjonsfilen ved bruk av nøkkelordet *info* og deretter tekst. Dette vil bli vist i informasjonsboksen når den

gitte noden er blitt klikket på. Figur 3.18 viser et eksempel på informasjon som vises i informasjonsboksen når en gitt node er blitt klikket på.



Figur 3.18: Informasjonsboks som viser informasjon etter at en node er blitt klikket på.

For å oppnå denne funksjonaliteten hentes informasjonen om noden ut fra *NodeInformation*-klassen, som blir laget når en konfigurasjonsfil blir lastet opp. I *MainPage.xaml.cs*-filen blir *NodeInformationService*-tjenesten brukt for å hente ut informasjon om en gitt node. Kode 3.8 viser hvordan *NodeInformationService*-tjenesten blir brukt og angir denne informasjonen i informasjonsboksen.

```
1 NodeInformation nodeInformation = ...
      _nodeInformationService.GetNodeInformationFromNode(...
      nodeId);
2 InfoText.Text = nodeInformation.Info;
```

Kode 3.8: Uthenting av informasjon om en gitt node.

3.7.5 Opplasting av CSV-fil

Når en konfigurasjonsfil har blitt lastet opp og en graf vises i applikasjonen, vil en ny knapp dukke opp i menyen. Figur 3.19 viser knappen som dukker opp etter en konfigurasjonsfil har blitt lastet opp. Knappen gjør det mulig å laste opp en CSV-fil som simulerer innsendte data til applikasjonen som beskrevet i kapittel 3.5.

Figur 3.19: Knapp i applikasjonen som muliggjør å laste opp CSV-fil.

Knappen blir laget i *MainPage.xaml*-filen, vist i kode 3.9. Her er *visibility*-verdien satt til *collapsed*. Dette betyr at knappen ikke vil være synlig i applikasjonen.

```
<Button x:Name="CSVFilePicker" Content="Choose CSV file"...
    Visibility="Collapsed"></Button>
```

Kode 3.9: Opprettelse av knapp for å velge CSV-fil.

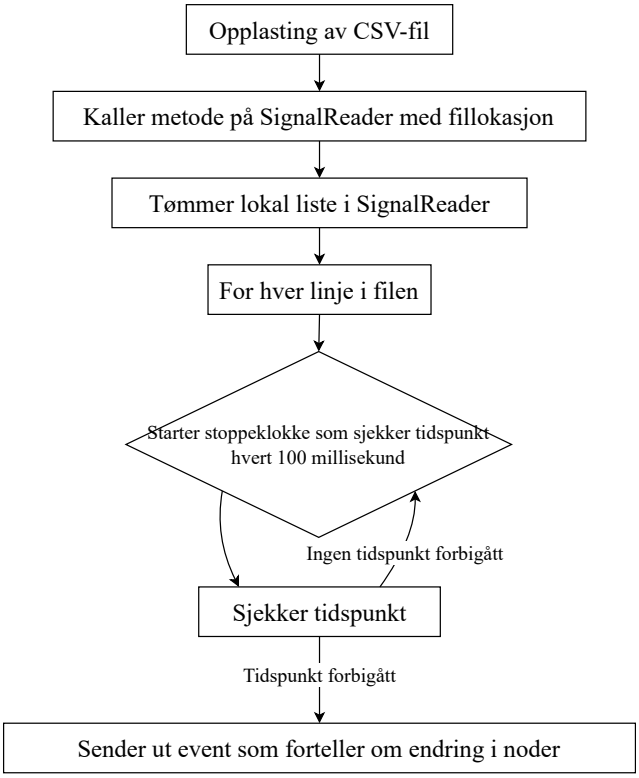
I filen *MainPage.xaml.cs* blir det lyttet på eventet som blir sendt ut når en konfigurasjonsfil blir valgt. Deretter blir *EnableCSVPicker*-metoden kalt, dersom eventet blir utløst. Metoden tar hånd om å gjøre knappen synlig. Deretter blir den valgte CSV-filen lest inn i applikasjonen. Metoden er vist i kode 3.10.

```
private void EnableCSVPicker(object? sender, Graph graph...
    )
{
    CSVFilePicker.Visibility = Visibility.Visible;
    FilePicker filePicker = new FilePicker();
    CSVFilePicker.Click += async (sender, e) =>
    {
        await filePicker.OnOpenFilePickerClickedAsync("....
            csv", (pickedFile) =>
        {
            if (pickedFile != null)
            {
                LoadTestFile(pickedFile.Path);
            }
        });
    };

}
```

Kode 3.10: Metoden *EnableCSVPicker*.

Figur 3.20 viser et flytskjema for prosessen av opplasting av en CSV-fil. Etter en CSV-fil har blitt lastet opp blir en metode i *SignalReader*-klassen med fillokasjonen på CSV-filen kalt. Deretter tømmer *SignalReader*-klassen en lokal liste og går gjennom hver linje i filen. I tillegg blir en stoppeklokke som sjekker tidspunkter i CSV-filen hvert 100 millisekund startet. Hvis tidspunktet ikke er forbigått, venter den nye 100 millisekunder. Hvis tidspunktet er forbigått, sender den ut et event som forteller om endring i signalet til relevante noder.

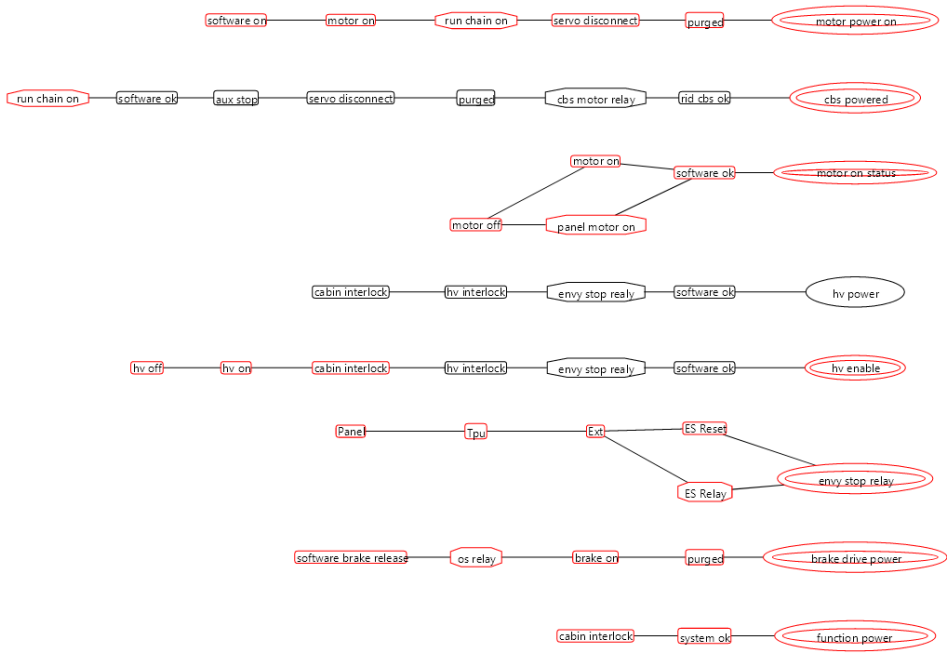


Figur 3.20: Flytskjema av prosessen for opplasting av CSV-fil.

3.7.6 Grafisk presentasjon av endring i signalene til sikkerhetstiltak

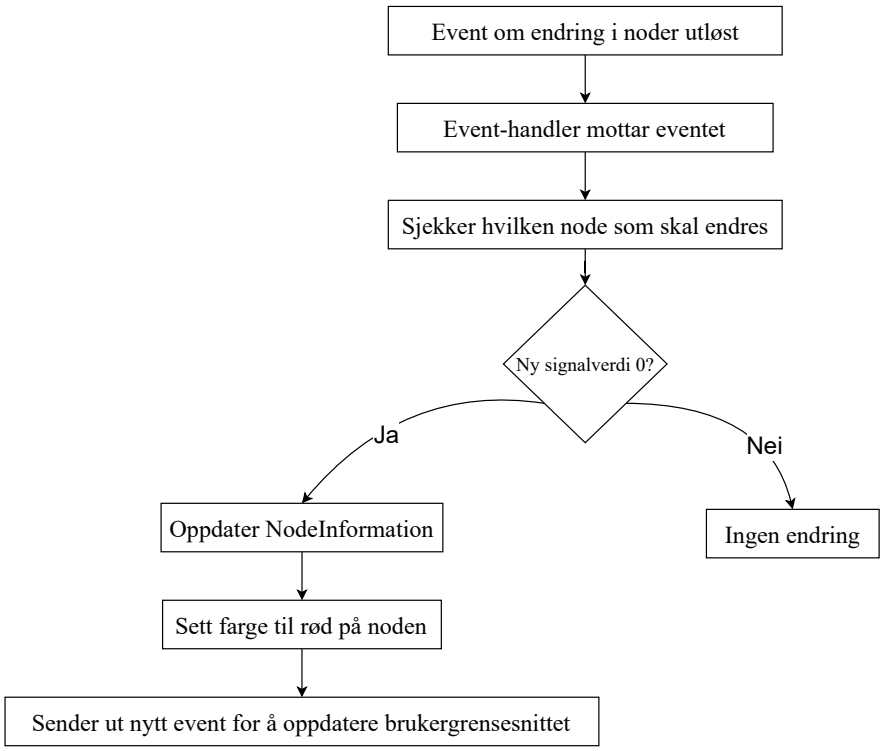
Signalet til hver node representerer tilstanden til sikkerhetstiltakene. Hvis signalet er 1, er det trygt å starte roboten. Hvis signalet er 0, er det ikke

trygt å starte roboten. Rød farge på noden representerer at signalet er 0. Figur 3.21 viser en graf med røde noder.



Figur 3.21: Grafen med signal 0 i ulike sikkerhetstiltak. Visualisert ved at nodene har rød farge.

Figur 3.22 viser et flytskjema over hva som skjer etter at et event har blitt utløst når en CSV-fil har blitt valgt. Først mottar en event-handler eventet og sjekker om den nye signalverdien er 0. Hvis den er 0, blir den relevante noden fra eventet hentet ut, fargen blir satt til rød og *NodeInformation*-klassen og brukergrensesnittet blir oppdatert. Hvis signalverdien ikke er 0, skjer det ingen endring.



Figur 3.22: Flytskjema for oppdatering av farge på noder.

Event-utløser og event-handler

Først blir eventet utløst ved bruk av nøkkelordet *Invoke* på *SignalChanged*, vist i kode 3.11. Dette utløser eventet og kaller funksjonen *OnSignal-Changed*

```
1 protected virtual void OnSignalChanged(...
      SignalChangedEventArgs e)
2 {
3      SignalChanged?.Invoke(this, e);
4 }
```

Kode 3.11: Event om endring i signalverdi utløst.

Kode 3.12 viser hva som skjer når event har blitt utløst. Først sjekkes det hvilken node endringen skal skje på. Deretter blir en annen funksjon kalt *SetSignal* med verdien på signalet kalt. Funksjonen *SetSignal* kaller deretter *SetColor*-funksjonen.

```
1 private void OnSignalChanged(object sender, ...
      SignalChangedEventArgs e)
2 {
3     if (e.Id == _id)
4     {
5         SetSignal(e.NewValue);
6     }
7 }
```

Kode 3.12: Event-handler for endring av signalverdi.

Signalverdi og endring i farge

Kode 3.13 viser *SetColor*-funksjonen. Denne funksjonen sjekker signalverdien og endrer deretter fargen på den gitte noden. Deretter utløser den et statisk event som sier at fargen har blitt endret. Dette vil oppdatere brukergrensesnittet.

```
1  internal async void SetColor(int signal)
2  {
3      Color previousColor = Node.Attr.Color;
4
5      if (signal == 0)
6      {
7          Node.Attr.Color = Color.Red;
8          if (Node.Attr.Color != previousColor)
9          {
10             ChangeShape(Shape.DoubleCircle);
11             // Raise the static event
12             ColorChanged?.Invoke();
13         }
14     }
15     if (signal == 1)
16     {
17         Node.Attr.Color = Color.Black;
18         if (Node.Attr.Color != previousColor)
19         {
```

```
20                 ChangeShape(Shape.Ellipse);
21                 // Raise the static event
22                 ColorChanged?.Invoke();
23             }
24         }
25 }
```

Kode 3.13: Endring av farge på node og utløsning av event for oppdatering av brukergrensesnitt.

3.7.7 Oppsummering av feil i en kjede

Den siste noden i en kjede fungerer som en oppsummering av kjeden. Denne noden vil ha fargen rød, dersom en node i kjeden også har fargen rød.

Ved å klikke på den siste noden, vil informasjon om hvilke noder som har fargen rød presenteres i informasjonsboksen. Figur 3.23 viser hvordan den siste noden i en kjede ser ut når den er blitt klikket på. Ved store konfigurasjonsfiler er dette nyttig, ettersom det kan være vanskelig å lokalisere hvor feilen har oppstått.

Figur 3.23: En gitt kjede hvor den siste noden i kjeden har blitt klikket på.

Det regnes ut et and-produkt for å sjekke om den siste noden skal ha fargen rød. And-produktet er basert på konjuksjon eller logisk-og. Konjuksjon er en måte å regne ut et sluttprodukt på, basert på to eller flere innverdier. For eksempel vil en innverdi på 0 og 1 gi sluttproduktet 0. Så lenge en av innverdiene er 0, vil alltid sluttproduktet bli 0. For at sluttproduktet skal bli 1, må alle innverdiene være 1. Dette er likt som med en kjede i grafen. Tabell 3.2 viser et eksempel på ulike innverdier og hva sluttproduktet blir ved å bruke logisk-og.

X	Y	=
0	0	0
0	1	0
1	0	0
1	1	1

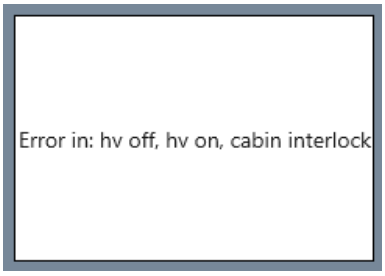
Tabell 3.2: Konjuksjon/logisk-og.

Kode 3.14 viser hvordan dette er løst i programmet. Her blir LINQ (Language Integrated Query) brukt for å hente ut verdiene til nodene. LINQ gjør det enklere å hente ut informasjon ved bruk av deres metoder. I koden sjekkes det om *Signal*-verdien til alle nodene er lik 1, utenom den siste noden. Hvis dette er sant setter den and-produktet til 1, noe som vil si at den siste noden i en kjede får verdien 1. Hvis dette ikke er sant, settes and-produktet til 0.

```
1 int andProduct = Nodes.Take(Nodes.Count - 1)
2                        .All(node => node.Signal == 1)
3                        ? 1
4                        : 0;
```

Kode 3.14: Utregning av and-produkt.

Når den siste noden i en kjede blir klikket på, vil eventuelle noder med farge rød bli oppført i informasjonsboksen. Dette er vist i figur 3.24, hvor tre noder med farge rød blir listet opp.



Figur 3.24: Informasjonsboks som lister opp alle noder i en gitt kjede som har farge rød, etter at den siste noden har blitt klikket på.

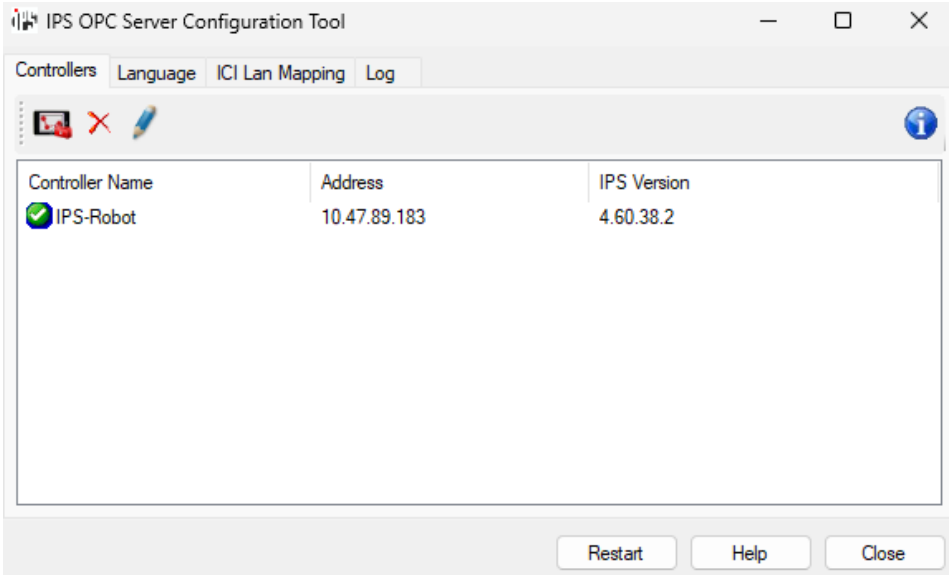
Kode 3.15 viser hvordan LINQ blir brukt til å hente ut de nodene som har signal verdi 0. Navnene til disse nodene blir satt sammen i en streng, og denne strengen blir satt som verdi i infoattributen til den siste noden i kjeden.

```
1  if (andProduct == 0)
2      {
3          var filteredLabelsList = Nodes
4              .Where(nodeInfo => nodeInfo.Signal == 0)
5              .Select(nodeInfo => nodeInfo.Node.LabelText)
6              .ToList();
7          if (filteredLabelsList.Any())
8          {
9              filteredLabelsList.RemoveAt(...
                   filteredLabelsList.Count - 1);
10         }
11         lastNode.Info = $"Error in: {string.Join(", ", ...
               filteredLabelsList)}";
```

Kode 3.15: Uthenting av noder med signalverdi 0.

3.7.8 Direkte signaler fra robot

I stedet for å laste opp en CSV-fil for innkommende signaler til ulike noder, er det mulig å koble seg opp til en lakkeringsrobot og hente ut direktesendte signaler. For å koble seg opp til en robot, brukes en OPC-server som nevnt i kapittel 2.6. ABB Bryne har selv utviklet en egen applikasjon som gjør det mulig å koble seg opp til lokale roboter på nettverket. Figur 3.25 viser et skjermbilde av applikasjonen med en robot koblet til.



Figur 3.25: Skjermbilde av ABB sin applikasjon som gjør det mulig å koble seg opp til lokale roboter på nettverket.

Samhandling med robot i .NET

For å samhandle med roboten i .NET brukes en NuGet-pakke ved navn *QuickOpc* som er utviklet av *OpcLabs* [25]. Denne pakken gjør det enkelt å hente ut informasjon fra OPC-serveren. Ved å innføre IP-adressen til roboten som er koblet til OPC-serveren i programmet, kan *QuickOPC*-pakken hente ut all nødvendig informasjon fra roboten. Dette inkluderer ulike noder, signaler og samtidig abonnere på eventer.

Koden som egen tråd i programmet

Når en konfigurasjonsfil har blitt lastet opp, kjører koden som samhandler med OPC-serveren som en egen tråd. Årsaken til at denne koden kjøres som en egen tråd, er at den kontinuerlig må lytte på eventer om oppdatering i signaler. Denne koden kan ikke kjøres et par ganger gjennom applikasjonens levetid, ettersom den da ikke ville fått meg seg alle endringer. Siden endringer i signaler kan skje når som helst, må koden lytte på eventer hele

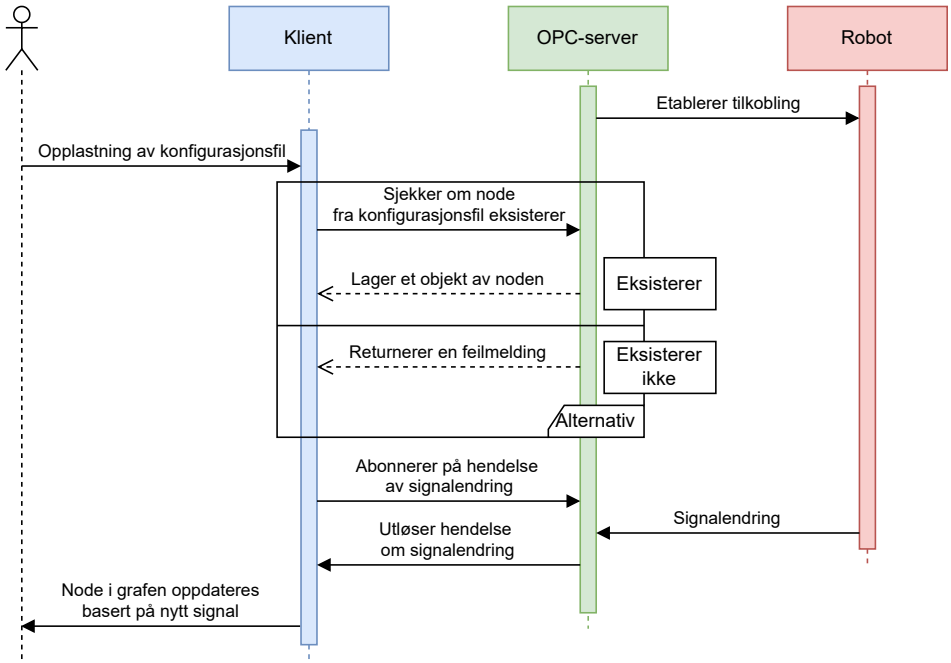
tiden. Etter at konfigurasjonsfilen har blitt lastet opp kjører applikasjonen koden som samhandler med OPC-serveren som en egen tråd. Dette er vist i kode 3.16.

```
1 _filePickerCreator.FilePicked += ((s, e) => Task.Run(() ...
      => ConnectToRobot()));
```

Kode 3.16: Egen tråd for kjøring av kode som samhandler med OPC-serveren.

Flyt mellom klient, OPC-server og robot

Først oppretter OPC-serveren en tilkobling med roboten. Denne tilkoblingen skjer i OPC-server-applikasjonen ABB har utviklet. Deretter opplaster brukeren en konfigurasjonsfil og klienten sjekker om nodenene eksisterer i OPC-serveren. Til slutt abonnerer klienten på eventer om signalendringer i OPC-serveren. Hvis en signalendring skjer fra roboten, sendes dette til OPC-serveren som igjen vil utløse et event om signalendring til klienten. Da vil nodenene i grafen oppdateres basert på det nye signalet og visualiseres for brukeren. Figur 3.26 viser et sekvensdiagram av flyten mellom klienten, OPC-serveren og roboten.



Figur 3.26: Sekvendsdiagram av flyten mellom klient, OPC-server og roboten ved opplastning av konfigurasjonsfil.

Uthenting av signaler fra robot

Koden som samhandler med OPC-serveren henter ut signaler fra roboten. Signalene fra roboten inkluderer en *id*, som identifiserer avsenderen av signalet. For at applikasjonen skal klare å hente ut signaler til riktige noder i grafen, er det viktig at nodene i konfigurasjonsfilen har den samme *id*'en som signalet fra roboten. Kode 3.17 viser uthenting av *id*'en til noder i konfigurasjonsfilen og leter deretter etter samme *id* i roboten gjennom OPC-serveren. Hvis den finner en *id* i roboten, legger den til en en representasjon av avsenderen, i form av et objekt av type *DAItemGroupArguments* til en liste.

```
1 List<DAItemGroupArguments> subscriptionArguments = new ...
      List<DAItemGroupArguments>();
2
3 foreach (var id in nodeInformationService....
```

```
    GetNodeInformationIdList())
{
    subscriptionArguments.Add(new DAItemGroupArguments("...
        ", "ABB.IPS.OPC.Server.DA.1",
    $"IPS-Robot.Nodes.10*47*89*183.Devices.Safety/{id}",
    200,
    state: id));
}
```

Kode 3.17: Uthenting av signaler fra OPC-serveren.

Direktesendte events fra robot

Etter at objektet har blitt lagt til i listen, lyttes det etter endringer i signaler. Når en endring i signalet fra roboten skjer, blir et event utløst. Kode 3.18 viser lyttingen på endring i signaler fra roboten gjennom OPC-serveren.

```
using (var client = new EasyDAClient())
{
    var eventHandler = new EasyDAItemChangedEventHandler...
        (client_ItemChanged);
    client.ItemChanged += eventHandler;

    int[] handleArray = client.SubscribeMultipleItems(...
        sAA);

    _shutdownEvent.WaitOne();
}
```

Kode 3.18: Lytting på endring i signaler fra robot.

Videresending av event til noder i grafen

Når eventet om endring i signal fra roboten blir utløst, blir kode 3.19 kalt. Denne koden sjekker verdien på signalet og utløser et event som nodene i programmet lytter på. Dette gjør at fargen på nodene endres. Det er også samme event som blir utløst når signalendring kommer fra opplastet CSV-fil.

```
1 static void client_ItemChanged(object sender, ...
      EasyDAItemChangedEventArgs e)
2 {
3     if (e.Succeeded) OnSignalChanged(new ...
          SignalChangedEventArgs((bool)e.Vtq.Value ? 1 : 0,...
           e.Arguments.State.ToString()));
4 }
```

Kode 3.19: Videresending av event fra robot til noder i grafen.

3.7.9 Skjule og vise menyen

Menyen kan skjules ved å trykke på ikonet øverst i applikasjonen, som vist i det røde området i figur 3.13. Dette resulterer i at menyen blir borte og hovedområdet hvor grafen vises blir større. Kode 3.20 er ansvarlig for å vise eller skjule menyen. Den kjører når ikonet øverst til venstre klikkes på. Dersom menyen ikke er skjult blir *Visibility* satt til *Collapsed* og menyen blir skjult. Hvis menyen er skjult blir *Visibility* satt til *Visible* og menyen dukker opp igjen.

```
1  if (!_toolBarHidden)
2  {
3      _toolBarHidden = true;
4      LeftArea.Visibility = Visibility.Collapsed;
5  }
6  else
7  {
8      _toolBarHidden = false;
9      LeftArea.Visibility = Visibility.Visible;
10 }
```

Kode 3.20: Bytte mellom visning av meny.

Kapittel 4

Diskusjon

Kapittelet tar for seg en drøfting rundt kvantitative verdier, valg av teknologi, drøfting av målene og videre utvikling av applikasjonen.

Hovedmålet med oppgaven var å utforske muligheten for å utvikle en forbedret applikasjon basert på Safety Chain Viewer i RobView, som muliggjør dynamisk konfigurasjon av sikkerhetstiltakene. Det var også et mål å utvikle en multiplattform-applikasjon. De neste underkapittelene vil med hovedmålene i baktanke diskutere sluttversjonen av applikasjonen samt testing av ytelsen og fremtidige forbedringer.

4.1 Datamaskin brukt for testing

En lokal datamaskin har blitt brukt for testing av applikasjonen. Denne brukes til vanlig i utvikling og testing av roboter, så det vil gi en god indikasjon på ytelsen til applikasjonen på en relevant datamaskin. Spesifikasjonene til datamaskinen er som følgende:

Intel Xeon Silver 4210 CPU 2.20GHz, 1 CPU, 16 GB RAM

Applikasjonen har ikke blitt testet på en håndholdt kontroller på grunn av tidsbegrensninger.

4.2 Metoder for testing

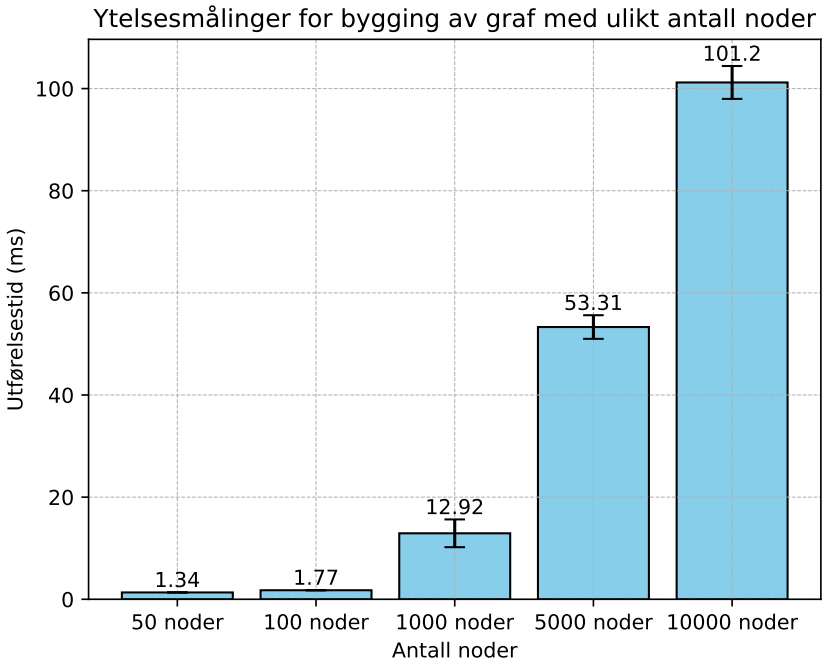
Ved testing av hvordan applikasjonen håndterer ulike oppgaver, ble det brukt to ulike metoder for testing av applikasjonen. Først ble tiden det tar å laste inn grafen med ulike antall noder tatt. Formålet med denne testen er å teste hvor godt applikasjonen håndterer å laste inn store konfigurasjonsfiler med mange sikkerhetstiltak. *BenchmarkDotNet* ble brukt for å utføre denne testen automatisk. Applikasjonen ble også testet ved å ta tiden det tok før brukergrensesnittet ble oppdatert. Denne testen ble gjort manuelt.

4.2.1 Manuell testing

Manuell testing ble utført for å teste ytelsen ved oppdatering av brukergrensesnittet. Ved oppdatering av brukergrensesnittet menes det når nodene i grafen endrer farge fra sort til rød. Manuell testing ble utført ved at tiden ble tatt med en stoppeklokke fra en CSV-fil ble lastet opp i applikasjonen, til alle nodene i grafen endret farge til rød. Meningen med testen var å sjekke hvor godt applikasjonen håndterte en endring i brukergrensesnittet, hvor alle nodene endret farge på samme tid. Ved bruk av manuell testing er det vanskelig å oppnå veldig nøyaktige resultater på grunn av menneskelige feil. Ettersom tiden det tok for applikasjonen å laste inn brukergrensesnittet gikk over flere sekunder, ses dette på som en akseptabel avrundingsfeil. Det var også nyttig å sjekke om applikasjonen hadde en grense på hvor mange noder den klarte å oppdatere samtidig, før programmet kræsjet.

4.3 Bygging av graf med ulikt antall noder

Den mest sentrale delen av applikasjonen er å bygge en graf når en konfigurasjonsfil blir lastet opp. For å opprettholde en god brukeropplevelse er det essensielt at dette skjer hurtig. Derfor ble tiden det tok fra en konfigurasjonsfil ble lastet opp til grafen ble bygget, testet ved bruk av BenchmarkDotNet. Figur 4.1 viser resultatet av denne testen.



Figur 4.1: Søylediagram som viser gjennomsnittlig utførelsestid for bygging av grafer med ulikt antall noder.

Figuren viser en trend hvor tiden det tar å bygge grafen øker med omtrent 10 ms for hver 1000. node konfigurasjonsfilen øker med. Dette stemmer overens med utviklingen av applikasjonen, ettersom tidskompleksiteten til algoritmen som bygger grafen er O(N). Dette betyr at kjøretiden vokser proporsjonalt med antall noder, ettersom den vokser med 10 ms for hver 1000. node.

4.3.1 Resultat med 50 og 100 noder

Ved opplasting av konfigurasjonsfil på henholdsvis 50 og 100 noder, viser resultatet at applikasjonen bruker i gjennomsnitt 1.34 ms og 1.77 ms på å bygge grafen. Sammenlignet med Safety Chain Viewer hvor det vanligvis finnes omtrent 100 noder i en graf, viser applikasjonen at den håndter basis-

tilfellet godt. I tillegg til at grafen bygges fort, gir applikasjonen en god brukeropplevelse hvor verktøyene knyttet til grafen fungerer godt. Det er ingen hakking eller treghet i programmet.

4.3.2 Resultat med 1000 noder

Ved en konfigurasjonsfil bestående av 1000 noder er det en betraktelig økning i utførselstid. Her bruker applikasjonen 12.92 ms, altså 0.0129 sekunder. Selv om det er en økning fra 100 noder, vil ikke brukeren merke en utførselstid på 0.01 sekund. Det utgjør altså ingen negativ innvirkning på brukeropplevelsen ved innlastning av konfigurasjonfil på denne størrelsen. Derimot oppleves applikasjonen dårligere enn med 100 noder, ettersom verktøyene som zooming og panning hakker til en viss grad. Det er vanskeligere å samhandle med grafen, noe som reduserer kvaliteten av brukeropplevelsen.

4.3.3 Resultat med 5000 noder

Ved en konfigurasjonsfil bestående av 5000 noder viser resultatet at applikasjonen bruker 53.31 ms (0.0531 s) på å bygge grafen. Her oppleves ventetiden betydelig lengere, men siden dette er en engangsjobb som skjer i starten av applikasjonen er det akseptabelt med noe innlastningstid. Den store forskjellen sammenlignet med mindre konfigurasjonsfiler er brukeropplevelsen av applikasjonen etter grafen er blitt lastet inn. Verktøyene zooming og panning hakker betydelig mer, og reduserer brukeropplevelsen drastisk. I helt nødvendige scenarioer er applikasjonen brukbar, men brukeropplevelsen vil oppleves som dårlig. Det er derfor ikke anbefalt med så store konfigurasjonsfiler.

4.3.4 Resultat med 10 000 noder

Ved en konfigurasjonsfil bestående av 10 000 noder viser resultatet at applikasjonen bruker 101.2 ms (0.1012 s) på å laste inn grafen. Applikasjonen håndterer byggingen av grafen, og ventetiden før grafen lastes inn er hva som kan forventes av en graf på denne størrelsen. Brukeropplevelsen med en

slik konfigurasjonsfil er såpass redusert at applikasjonen oppleves som ikke brukbar.

4.3.5 Grense på opplastning av konfigurasjonsfil

Teoretisk sett er det ingen grense på hvor stor graf MsAGL kan håndtere. Derimot er ikke MsAGL designet for å håndtere veldig store grafer og MsAGL har blitt testet ved å bygge en graf med 15 000 koblinger. Da ble en datamaskin med 4 GB ram brukt og tiden det tok å laste inn grafen var på omtrent 10 minutter. [23] Dette er derimot alt for lang tid skal det opprettholdes en god brukeropplevelse i applikasjonen.

Ut ifra testene er grensen for en behagelig brukeropplevelse 1000 noder. Her laster grafen umiddelbart inn og verktøyene zooming og panning fungerer uten noen problemer. Applikasjonen er derimot brukbar opptil 5000 noder, men brukeropplevelsen er såpass redusert at dette ikke er å anbefale.

Dette resultatet er spesifikt for akkurat den datamaskinen testene ble gjennomført på. Ved bruk av andre datamaskiner som har bedre spesifikasjoner, kan det være mulig at resultatene viser noe annet og at applikasjonen håndterer større konfigurasjonsfiler. I tillegg vil det også utvikles bedre prosessorer og minne på datamaskinene som blir laget over tid, noe som vil gjøre at vanlige datamaskiner kan håndtere applikasjonen bedre i fremtiden.

Derimot er datamaskinen som testene ble utført på relevant. Denne datamaskinen blir til vanlig brukt for testing og utvikling av lakkeringsroboter. Ytelsen til applikasjonen på denne datamaskinen er dermed betydningsfull, ettersom det er en slik datamaskin applikasjonen vil bli brukt på hvis den blir tatt i bruk av ABB.

4.4 Testing av endring i farge på noder

For å forstå hvordan applikasjonen håndterer vanlig bruk er det hensiktsmessig å utforske ytterpunktene av endring i grafen. At et sikkerhetstiltak endrer tilstand, altså at noden endrer farge er den mest datakrevende ope-

rasjonen i applikasjonen. Årsaken til dette er at hele grafen må lastes inn på nytt for å oppdatere det grafiske brukergrensesnittet. Ved å se på ulike scenarioer hvor alle nodene endrer seg samtidig, kan en maksgrense for applikasjonen etableres.

4.4.1 Sammendrag av resultater

Tabell 4.1 viser et sammendrag av tiden det tok å endre farge på de ulike nodene. Testingen ble gjort manuelt ved å laste inn CSV-filer som simulerer innsendte data, som beskrevet i kapittel 4.2.1

Tabell 4.1: Resultat av manuell testing på endring i farge på noder. Tabellen viser antall noder, gjennomsnittstid, om applikasjonen kræsjer og om applikasjonen henger.

Antall noder	Tid i sekunder (gjennomsnitt)	Kræsj	Henger seg
100	Umiddelbart	Nei	Nei
500	6.78	Nei	Ja
980	38.01	Ja	Ja

4.4.2 Analyse av resultater

Gjennomsnittstiden på oppdatering av brukergrensesnittet ved forskjellige antall noder, ble regnet ut av tiden ved 10 uavhengige forsøk. 10 forsøk ble gjennomført for å kvalitetssikre resultatet. Derimot kunne det vært hensiktsmessige å gjennomføre forsøket flere enn 10 ganger, for å få et enda mer nøyaktig resultat. Ettersom et forsøk på eksempelvis 980 noder tok i gjennomsnitt 38.01 sekunder, ble et kompromiss på antall forsøk gjort, med tanke på tidsbruk.

100 noder

Resultatet viser at ved en kollektivt feiling på 100 noder, håndterer applikasjonen det bra. Oppdateringen i brukergrensesnittet skjer umiddelbart. Umiddelbart i denne konteksten betyr så fort at det ikke var mulig å ta

tiden manuelt. Applikasjonen har heller ingen andre problemer med 100 noder, ettersom den ikke kræsjer eller henger seg. At applikasjonen henger seg, betyr at programmet fryser og ikke er mulig å bruke imens det laster.

500 noder

Ved 500 noder viser resultatet at applikasjonen bruker i gjennomsnitt 6.78 s. Dette er en betydelig økning fra 100 noder og påvirker brukeropplevelsen negativt. I tillegg til en økning i innlastningstid, henger programmet seg mens det oppdaterer brukergrensesnittet. Dette betyr at applikasjonen ikke fungerer å samhandle med. Etter gjennomsnittlig 6.78 s, fungerer dermed applikasjonen slik den skal.

980 noder

Ved 980 noder viser resultatet at applikasjonen bruker i gjennomsnitt 38.01 s. I tillegg til at applikasjonen bruker lang tid på å laste inn endringene i brukergrensesnittet, kræsjer programmet 4 av 10 ganger. Det er vanskelig å vite den eksakte årsken til at applikasjonen kræsjer, men følgende faktorer kan være årsaken:

- **Minneforbruk**: Store grafer krever betydelig med minne for å kunne lagre informasjon om grafen og dens innhold. Når alle nodene på grafen skal oppdateres på samme tid, vil det føre til økt bruk av minne. Hvis datamaskinen ikke har nok ledig minne til å håndtere denne operasjonen, kan det føre til at applikasjonen kræsjer.
- **Visning av grafen**: Det kan være problematisk for MsAGL-biblioteket å vise store grafer i applikasjonen. Når hver node skal oppdateres krever det mye fra MsAGL. Å oppdatere hele brukergrensesnittet kan føre til at MsAGL ikke klarer å håndtere en så stor oppdatering på samme tid.
- **Overbelastning på CPU**: Ved å endre tilstanden på alle nodene i grafen kan det føre til at applikasjonen krever ekstra mye CPU-kraft. Dette kan føre til at denne handlingen blir for kompleks for

CPU'en. Hvis applikasjonen fryser for lenge vil operativsystemet til slutt avslutte denne operasjonen, noe som fører til at applikasjonen kræsjer.

Når applikasjonen kræsjer 4 av 10 ganger konkluderes det med at grensen for maksimalt antall noder som kan oppdateres på samme tid er 980.

Konklusjon av testresultatene med endring i farge på noder

At alle nodene endrer farge samtidig er et ytterpunkt som er lite sannsynlig vil oppstå ved vanlig bruk av applikasjonen. Derimot er ikke nodene helt uavhengige av hverandre, noe som vil si at hvis en node endrer farge, kan det føre til at en annen node også endrer farge. For eksempel ved endring av modus en robot er i. I tillegg fungerer den siste noden i en kjede som en oppsummering, noe som vil si at den endrer farge ut ifra om nodene i kjeden endrer farge. Applikasjonen vil også få statusen til alle sikkerhetstiltakene i starten av levetiden. Dette kan føre til en eventuell oppdatering av mange noder på samme tid. Denne initialiseringen er kritisk for å representere tilstanden til sikkerhetstiltakene korrekt fra begynnelsen. Selv om en oppdatering av alle nodene i applikasjonen er et ytterpunkt som sjeldent vil inntreffe, er det scenarioer hvor mange noder kan endre farge. Det er derfor kritisk å vite om denne grensen og hvor mange sikkerhetstiltak en robot har, før man tar i bruk applikasjonen.

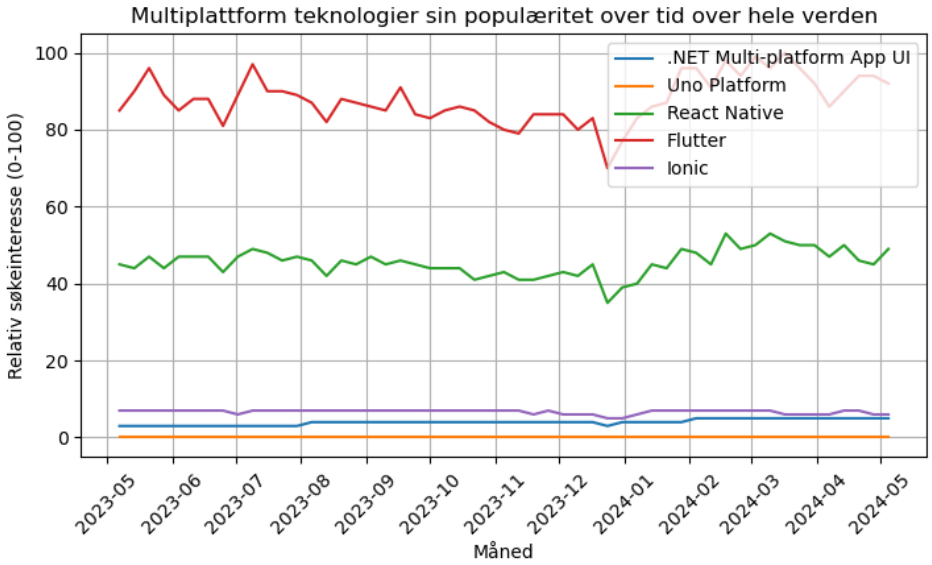
Det er vanskelig å si hvor mange sikkerhetstiltak en robot har, ettersom det vil variere fra robot til robot. Safety Chain Viewer i RobView viser rundt 100 noder i den grafiske visningen. Applikasjonen kan dermed håndtere en del mer enn Safety Chain Viewer i dag. Skulle det være behov for større representasjoner av sikkerhetstiltakene, vil det kreve mer utvikling av applikasjonen. Noen av tiltakene som kan utvikle applikasjonen videre, er beskrevet i kapittel 4.7.

4.5 Valg av teknologi

Valg av riktig teknologi er essensielt ved utviklingen av en applikasjon. I denne delen drøftes det rundt valgene tatt under utviklingen og hvordan bruken av den valgte teknologien har påvirket sluttresultatet.

4.5.1 UNO Platform

Valg av teknologi for å lage multiplattform-applikasjoner starter ved valg av kodespråk. Det var i ABB Bryne sitt ønske å bruke C#, ettersom dette er kodespråket RobView er utviklet med. Microsoft har et multiplattform utviklingsverktøy kalt .NET Multi-platform App UI (MAUI). Derimot støtter ikke MAUI WebAssembly og applikasjonen ville dermed ikke hatt muligheten til å kunne kjøres i nettleseren [18]. Grunnet forutsetningen for oppgaven om Windows og WebAssembly støtte var UNO Platfrom derfor et hensiktsmessig valg. Derimot kom valget med mindre dokumentasjon og et mindre felleskap, noe som gjør det vanskeligere å finne frem til løsninger. UNO har en relativt liten markedsandel i forhold til MAUI og andre multiplattformutviklingsverktøy som Flutter, React Native og Ionic. Figur 4.2 viser den relative søkehistorikken på Google mellom de ulike nevnte multipltattformutviklingsverktøyene.



Figur 4.2: Linjediagram som viser den relative søkehistorikken mellom ulike multiplattform-utviklingsverktøy. Datakilde: Google Trends (`https://www.google.com/trends`)

Selv om UNO Platform har lite populæritet sammenlignet med andre multiplattform-utviklingsverktøy, har UNO fungert bra for utviklingen av applikasjonen. Ettersom applikasjonen ble utviklet med C# og .NET i samarbeid med UNO, var det ingen åpenbare negative konsekvenser med UNO.

4.5.2 Microsoft Automatic Graph Layout (MsAGL)

Microsoft Automatic Graph Layout (MsAGL) gjorde det enkelt å automatisk bygge en graf ut ifra en konfigurasjonsfil, som var hovedmålet med oppgaven. Derimot var det ikke mulig å vise grafen i applikasjonen med MsAGL i kombinasjon med UNO Platform. Dette skapte problemer for utviklingen av applikasjonen og løsningen var en egen NuGet-pakke utviklet på ABB, som beskrevet i kapittel 2.4.3. Uten denne NuGet-pakken ville det ikke vært mulig å vise grafen bygget av MsAGL i en UNO-applikasjon. Gruppen slet med å finne andre NuGet-pakker som var kompatible med UNO for å fremvise grafer visuelt. Alternativt kunne grafen blitt tegnet

med et grafikkbibliotek som SkiaSharp. Derimot ville dette ført til en stor økning i mengden kode for å bygge grafen, ettersom man måtte tegnet hver node manuelt. På den andre siden kunne dette ført til mer kontroll over designet og andre symboler på nodene. Ved å velge MsAGL ble resultatet en hurtigere utviklingsfase på bekostning av tilpasningsmuligheter.

4.5.3 OPC Classic

Ettersom ABB bruker en OPC-server som kun kan kommunisere med en OPC Classic klient, var det ikke noe annet valg enn å bruke OPC Classic. Problemet med OPC Classic er at det er en gammel teknologi og ble byttet ut med OPC UA i 2008. Det er dermed et lite fellskap rundt teknologien og lite med dokumentasjon. Det var også et problem å finne en NuGet-pakke som gjorde det mulig å kommunisere med OPC-serveren med .NET. Utenom vanskeligheter med utviklingen, er det problematisk å bruke en gammel teknologi. Årsaken til dette er blant annet:

- **Sikkerhetsrisiko**: Utdatert teknologi får ikke oppdateringer basert på risiko. Det vil derfor være problematisk å bruke en utdatert teknologi over tid, ettersom det blir utviklet nye måter å angripe på
- **Kompatibilitetsproblemer**: Gammel teknologi kan føre til dårligere kompatibilitet med andre teknologier, operativsystemer og enhter. Dette ble erfart ettersom det ikke fungerte å kjøre applikasjonen i nettleseren med OPC Classic
- **Manglende funksjonalitet**: Utdatert teknologi kan mangle viktige funksjonaliter som ikke vil bli innført på grunn av manglende oppdateringer
- **Kostnader ved vedlikehold**: Kostnadene ved vedlikhold kan øke ettersom det krever spesialkunnskap å håndtere utdatert teknologi

Etter tilkoblingen med OPC-serveren ble satt opp korrekt i applikasjonen, var det rett frem å hente ut data fra lakkeringsroboten og vise det i applikasjonen.

4.6 Problemer ved kjøring av applikasjon i nettleseren

Applikasjonen møtte på problemer når den skulle kjøres i nettleseren. Under vil årsakene til dette bli beskrevet.

4.6.1 MsAGL med UNO Platform

Et av målene med oppgaven var å utvikle en multiplattform-applikasjon som kan kjøre på Windows og i nettleseren. Ved å kombinere MsAGL og UNO Platform kjører applikasjonen i nettleseren, men klarer ikke å bygge grafen.

Hypotesen for dette problemet er at det er en forskjell i NuGet-pakken utviklet på ABB og NuGet-pakkene UNO er utviklet på. NuGet-pakken som er utviklet på ABB, bruker en nyere versjon av SkiaSharp enn det UNO bruker for å vise grafen i applikasjonen. Dette er problematisk ettersom UNO trenger SkiaSharp for å vise grafen i nettleseren.

Siden applikasjonen klarer å vise grafen på Windows, skal problemet med visning av grafen i nettleseren automatisk fikses i fremtiden. Det skal ikke være behov å endre noe på koden, ettersom samme kodebase skal fungere på alle plattformer. Hvis UNO oppdaterer til samme SkiaSharp versjon, skal det automatisk fungere å vise grafen i nettleseren.

4.6.2 OPC Classic med UNO Platform

Når samhandling med robotene ble integrert, kunne ikke applikasjonen kjøres i nettleseren lenger. Årsaken til dette er at OPC-serveren fra ABB bruker OPC Classic. OPC Classic er ikke designet for å brukes i nettleseren og fungerer kun på Windows-spesifikke applikasjoner.

For å løse dette problemet må OPC UA benyttes i stedet for OPC Classic. OPC UA er designet for å være plattformuavhengig og støtter kommunikasjonsprotokoller som brukes i nettlesere. For at applikasjonen skal bruke

OPC UA må også OPC-serveren til ABB bruke OPC UA.

En overgang til OPC UA har også sine konsekvenser. For det første må endringer i koden til applikasjonen utføres for å tilrettelegge for OPC UA-kommunikasjon. I tillegg må en endring i ABB sin OPC-server gjøres, slik at den støtter OPC UA. Dette kan innebære både tids- og ressurskrevende prosesser og er utenfor gruppens kompetanse og mål for oppgaven.

4.7 Fremtidig utvikling av applikasjonen

Hovedmålet med applikasjonen, som var å dynamisk bygge en graf basert på en konfigurasjonsfil, er blitt oppnådd. Derimot har ikke målet med at applikasjonen skal kjøres på Windows og i nettleseren blitt fullstendig oppnådd. Det er derfor rom for forbedring og videre utvikling av applikasjonen. Noen forslag er diskutert i følgende underkapitler.

4.7.1 Visning av graf i nettleseren

Dersom UNO Platform ikke oppgraderer til den samme versjonen av SkiaSharp som blir brukt i applikasjonen, kan det være hensiktsmessig å finne en annen teknologi for å vise grafen.

Direkte bruk av SkiaSharp

I stedet for å bruke MsAGL-biblioteket kan SkiaSharp brukes direkte i applikasjonen. Fordelen ved bruk av SkiaSharp er at det er kompitabelt med flere plattformer enn MsAGL. I tillegg ville applikasjonen kunne kjøres i nettleseren ved bruk av SkiaSharp. [28]

Ulempen ved direkte bruk av SkiaSharp er at det krever betydelig mer utvikling. Ved bruk av SkiaSharp må grafen lages manuelt, i stedet for å bruke nøkkelord som *graph*, *nodes* og *edges* som er tilgjengelig i MsAGL.

Bruk av andre grafverktøy

Det finnes også andre alternativer for å bygge grafer i nettleseren. Eksempelvis Graphviz eller D3.js. Dette er kraftige grafverktøy som muliggjør visualisering av store datasett. Derimot er det viktig å forsikre kompatibilitet med UNO Platform skal dette implementeres i applikasjonen. Det vil også kreve betydelig med videre utvikling, skal selve grafverktøyet byttes ut, ettersom det er hovedfunksjonen med applikasjonen.

4.7.2 Endring av kommunikasjonsprotokoll

Som nevnt tidligere er OPC Classic en av årsakene til at applikasjonen ikke kan kjøre i nettleseren med WebAssembly. I en eventuelt videreføring av applikasjonen bør andre kommunikasjonsprotokoller med roboten som kan kjøre på WebAssembly utforskes. Oppgraderer ABB OPC-serveren til å benytte OPC UA, trenger kun applikasjonen å oppgradere fra OPC Classic til OPC UA. Koden er satt opp slik at det vil være lett å bytte til en ny kommunikasjonsprotokoll og utløse et event for å fortelle nodene om endring.

4.7.3 Automatisk opplastning av konfigurasjonsfil

En svakhet ved applikasjonen er at det kreves mye kunnskap om roboten for å sette opp en konfigurasjonsfil. Årsaken til dette er at *id*'en på nodene må være den samme som det sikkerhetstiltaket heter i roboten. Et naturlig steg videre vil være at roboten sender over en kofigurasjonsfil av sine sikkerhetstiltak til applikasjonen ved oppstart. Dette vil sikre at applikasjonen alltid viser den riktige konfigurasjonen istedet for at en bruker må oppdatere konfigurasjonsfilen.

4.7.4 Bedre brukeropplevelse ved større konfigurasjonsfiler

Ved scenarioer der det er nødvendig å laste opp veldig store konfigurasjonsfiler, kan brukeropplevelsen bedres. Applikasjonen håndterer godt 1000 noder og det er langt fler enn antall sikkerhetstiltak på en vanlig robot i dag.

Skal det derimot være behov for mer enn det i fremtiden, må applikasjonen utvikles videre for å håndtere større konfigurasjonsfiler.

4.7.5 Bedre respons ved feiling i flere noder

Noe som hadde gjort applikasjonen mer responsiv ved feiling i flere noder samtidig, hadde vært at den ikke oppdaterer det grafiske brukergrensenittet for hver node som ble endret. For å oppdatere det grafiske brukergensesnittet må hele grafen lastes inn på nytt i applikasjonen, noe som er en tidskrevene operasjon. Problemet med applikasjonen som den er nå er at grafen lastes inn på nytt for hver node som endrer seg. Det finnes flere løsninger på dette problemet. Følgende løsninger kan implementeres ved fremtidig utvikling av applikasjonen.

Batch-oppdateringer

I stedet for å oppdatere brukergrensesnittet for hver endring i en node, kan applikasjonen samle flere endringer og utføre en enkelt oppdatering av brukergrensesnittet. Et altnerativ er å utføre tidsbestemte oppdateringer. Da kan applikasjonen samle opp alle nodeendringer innenfor et bestemt tidsvindu også oppdatere brukergrensesnittet en gang for alle endringene. En annen måte er å bruke event-basert oppdatering. Da vil applikasjonen vente til et visst antall endringer har skjedd og deretter oppdatere brukergrensesnittet.

Sammenligning av tilstander

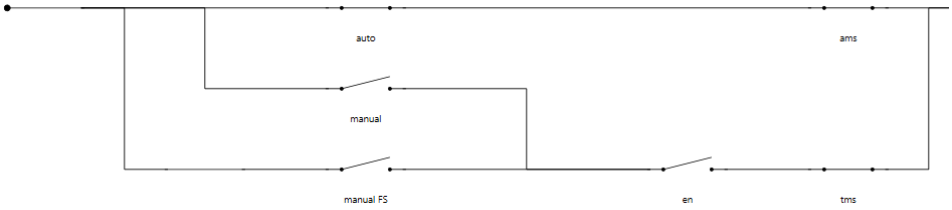
En annen måte å redusere oppdateringene er å sammenligne tilstanden av grafen og oppdatere brukergrensesnittet deretter. En algoritme for dette kan være som følgende:

- Lagre en kopi av grafstrukturen i minnet
- Sammenlign den nåværende grafstrukturen med den som er lagret i minne

- Beregn en minimal mengde endringer som må skje
- Oppdater kun de delene av grafen som har endret seg

4.7.6 Noder i parallell ved valg av forskjellige moduser

Safety Chain Viewer har muligheten til å endre symbolet til en node i visningen. For eksempel har en lakkeringsrobot flere moduser, hvor den kan kjøres i enten manuell- eller auto-modus. Ved fremtidig utvikling av applikasjonen bør funksjonalitet hvor symbolene kan endre seg, avhengig av tilstanden til sikkerhetstiltakene implementeres. I tillegg vises de ulike modusene parallelt i Safety Chain Viewer. Figur 4.3 viser ulike noder som går i parallell. Her er auto-modusen aktivert og vises ved at symbolet har en rett linje. Manuell-modusen er ikke aktivert og vises ved at linjen ikke er tilkoblet resten av kjeden. Hvis modusen på roboten endrer seg, vil også symbolene på modusene i grafen endre seg.



Figur 4.3: Skjermbilde fra Safety Chain Viewer i RobView som viser flere noder som går i parallell.

I applikasjonen er denne funksjonaliteten ikke implementert på grunn av tidsbegrensninger. Ved fremtidig utvikling bør muligheten til å definere noder som går i parallell i konfigurasjonsfilen implementeres. Dette kan oppnås ved å ha flere nøkkelord i konfigurasjonsfilen som sier hvilke noder som er tilkoblet en parallell kjede. I tillegg bør applikasjonen ha muligheten til å endre på symbolene, avhengig av tilstanden på sikkerhetstiltakene. Det er i dag kun mulighet til å endre på fargen på nodene.

En slik funksjonalitet kan implementeres ved bruk av MsAGL. Derimot ble det testet å endre symbolene under utviklingen av applikasjonen, men viste seg å være utfordrende. Det var problematisk å designe egne noder og det

vil kreve en overskriving av MsAGL-biblioteket, ettersom det ikke støtter endring av symbolene ut av boksen. Det vil derfor kreve mer utvikling for å endre på symbolene som kommer med MsAGL eller så må andre grafverktøy som støtter andre symboler utforskes.

4.7.7 Bedre design

I utviklingen av applikasjonen har design ikke vært prioritert, ettersom funksjonalitet har vært det viktigste. Fokuset gjennom utviklingen av applikasjon har dermed vært å sikre at applikasjonen fungerer slik at den skal og at de grunnleggende funksjonalitetene er på plass. Ved videre utvikling kan det dermed være hensiktsmessig å fokusere på designet av applikasjonen.

Bedre design kan føre til lettere navigasjon i applikasjonen samt en bedre brukeropplevelse. Et godt design påvirker hvor brukervennlig applikasjonen er. Det er dermed essensielt med et godt brukergrensesnitt som kan øke brukertilfredsheten og redusere læringskurven for brukeren av applikasjonen.

Ved fremtidig utvikling bør fokuset være på brukergrensesnittet og teste hvordan applikasjonen oppfører seg på ulike enheter. Siden ABB bruker håndholdte kontrollere i tillegg til datamaskiner ved samhandling med robotene, kan det være hensiktsmessig å teste at brukergrensesnittet oppfører seg slik det skal på en håndholdt kontroller.

Bedre design kan føre til økt adoptasjon av applikasjonen samt reduserte supportkostnader. En intuitiv applikasjon kan føre til enklere bruk og reduserte kostnader knyttet til opplæring og vedlikehold.

Kapittel 5

Konklusjon

Målet med oppgaven var å utforske muligheten til å utvikle en forbedret versjon av et delprogram i RobView, nærmere bestemt Safety Chain Viewer. Spesifikt skulle den grafiske visningen av sikkerhetstiltakene konfigureres dynamisk ved opplastning av en konfigurasjonsfil. Dette målet ble nådd og ønsket funksjonalitet i applikasjonen ble utviklet.

Applikasjonen hadde derimot noen begrensninger ved størrelsen på grafen. Ut ifra resultatene håndterer applikasjonen maksimalt 1000 noder i en konfigurasjonsfil for å opprettholde god brukeropplevelse. Ved endring i tilstand på nodene, var grensen 980 noder ettersom applikasjonen kræsjet ved flere noder enn dette.

Målet om å utvikle en multiplattform-applikasjon ble ikke nådd. Applikasjonen har mulighet til å kjøre på Windows, men ikke i nettleseren, grunnet begrensninger i den brukte teknologien. Kombinasjonen mellom MsAGL og UNO Platform gjør det ikke mulig å bygge grafen i nettleseren, på tidspunktet av denne oppgaven. I tillegg førte den utdaterte teknologien OPC Classic til at applikasjonen ikke kan kjøre i nettleseren.

Det er derfor et stort potensiale for videre utvikling av applikasjonen. Applikasjonen kan både yte bedre, samt implementere mer funksjonalitet som er tilgjengelig i RobView i dag og kjøre på flere plattformer enn bare Windows.

Bibliografi

[1] Maskinforordningen. https://www.regjeringen.no/no/sub/eos-notatbasen/notatene/2022/okt/forslag-til-maskinforordning/id2951107/. Sist besøkt: 10. mai, 2024.

[2] ABB. 50 år siden jærbuenes banebrytende oppfinnelse av lakkeringsroboten. https://new.abb.com/news/no/detail/36667/50-ar-siden-jaerbuenes-banebrytende-oppfinnelse-av-lakkeringsroboten. Sist besøkt: 23. januar, 2024.

[3] ABB. Om oss. https://new.abb.com/no/om-oss. Sist besøkt: 23. januar, 2024.

[4] Andrey Akinshin. Benchmarkdotnet. https://github.com/dotnet/BenchmarkDotNet. Sist besøkt: 6. mai, 2024.

[5] Don Box and Chris Sells. *Essential. Net: the common language runtime*, volume 1. Addison-Wesley Professional, 2003.

[6] Forbes. Abb. https://www.forbes.com/companies/abb/?sh=5b1ab939520e. Sist besøkt: 7. mai, 2024.

[7] OPC Foundation. Classic. https://opcfoundation.org/about/opc-technologies/opc-classic/. Sist besøkt: 2. april, 2024.

[8] OPC Foundation. Unified architecture. https://opcfoundation.org/about/opc-technologies/opc-ua/. Sist besøkt: 2. april, 2024.

[9] GitHub. About continuous integration. https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration. Sist besøkt: 16. februar, 2024.

[10] Google. About skia. https://skia.org/about/. Sist besøkt: 7. mai, 2024.

[11] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, 2004.

[12] Haldor Hove. Robottyper. https://ndla.no/article/35720. Sist besøkt: 10. mai, 2024.

[13] Microsoft. Architectural principles: Dependency inversion. https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion. Sist besøkt: 20. februar, 2024.

[14] Microsoft. Introduction to net. https://learn.microsoft.com/en-us/dotnet/core/introduction. Sist besøkt 30. januar, 2024.

[15] Microsoft. An introduction to nuget. https://learn.microsoft.com/en-us/nuget/what-is-nuget. Sist besøkt: 7. mai, 2024.

[16] Microsoft. .net dependency injection. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. Sist besøkt: 20. februar, 2024.

[17] Microsoft. A tour of the c language. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. Sist besøkt: 8. mars, 2024.

[18] Microsoft. What is .net maui? https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-8.0. Sist besøkt: 9. mai, 2024.

[19] Microsoft. Windows ui library (winui). https://learn.microsoft.com/en-us/windows/apps/winui/. Sist besøkt: 30. januar, 2024.

[20] Microsoft. Worker services in .net. https://learn.microsoft.com/en-us/dotnet/core/extensions/workers. Sist besøkt: 20. februar, 2024.

[21] Microsoft. Xaml overview (wpf .net). https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-8.0. Sist besøkt: 23. januar, 2024.

[22] Lev Nachmanson. Microsoft automatic graph layout. https://github.com/microsoft/automatic-graph-layout. Sist besøkt: 6. februar, 2024.

[23] Lev Nachmanson. Microsoft automatic graph layout. https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/. Sist besøkt: 13. mai, 2024.

[24] Nikola S. Nikolov. Sugiyama algorithm, 2016.

[25] OpcLabs. Opc client toolkit made right. https://www.opclabs.com/products/quickopc?nuget. Sist besøkt: 19. april, 2024.

[26] UNO Platform. How it works. https://platform.uno/how-it-works/. Sist besøkt: 8. mai, 2024.

[27] UNO Platform. How uno platform works. https://platform.uno/docs/articles/how-uno-works.html. Sist besøkt: 30. januar, 2024.

[28] UNO Platform. Skiasharp support for webassembly via uno platform. https://platform.uno/blog/skiasharp-support-for-webassembly-via-uno-platform/. Sist besøkt: 13. mai, 2024.

[29] W3Schools. Introduction to xml. https://www.w3schools.com/xml/xml_whatis.asp. Sist besøkt: 6. februar, 2024.

[30] World Wide Web Consortium. Webassembly. https://webassembly.org/. Sist besøkt: 2. februar, 2024.

Figurer

Tabeller

Vedlegg A

Dynamisk opprettelse av konfigurasjonsfil ved hjelp av Python.

```
1  import random
2  from xml.etree.ElementTree import Element, SubElement, ...
       tostring, Comment
3
4  from xml.dom.minidom import parseString
5
6  def pretty_xml(element):
7      # Formaterer XML-koden
8      rough_string = tostring(element, 'utf-8')
9      reparsed = parseString(rough_string)
10     return reparsed.toprettyxml(indent="  ")
11
12 def create_graph(num_chains, labels, shapes):
13     graph = Element('graph')
14     nodes = SubElement(graph, 'nodes')
15     edges = SubElement(graph, 'edges')
16
17     node_id = 1
18
19     for chain in range(num_chains):
20         first_node = True
21         last_node = False
22
```

```
23          # Setter antall noder per kjede
24          nodes_per_chain = random.randint(80, 120)
25
26          for node in range(nodes_per_chain):
27              if node == nodes_per_chain - 1:
28                  last_node = True
29                  # Setter symbolet til siste node i en ...
                        kjede til 'ellipse'
30                  shape = 'ellipse'
31              else:
32                  # Setter symbolet til andre noder til et...
                        tilfeldig symbol
33                  shape = random.choice(shapes)
34
35              # Setter en tilfeldig 'label'
36              label = random.choice(labels)
37
38              # Lager node-elementet
39              node_element = SubElement(nodes, 'node', id=...
                    str(node_id), label=label, shape=shape)
40
41              if first_node:
42                  # Setter 'firstnode' til 'true' hvis det...
                        er forste node i en kjede
43                  node_element.set('firstnode', 'true')
44                  first_node = False
45              if last_node:
46                  # Setter 'lastnode' til 'true' hvis det ...
                        er siste node i en kjede
47                  node_element.set('lastnode', 'true')
48
49              if node > 0:
50                  # Lager edge-elementet
51                  edge_element = SubElement(edges, 'edge',...
                        source=str(node_id - 1), target=str(...
                        node_id))
52
53              node_id += 1
54
55      return pretty_xml(graph)
56
57  # Liste med forskjellige 'labels' noder kan ha
58  labels = ["Panel", "Tpu", "Ext", "ES Reset", "ES Relay",...
        "envy stop relay", "motor off", "motor on", "panel ...
       motor on", "software ok", "motor on status", "...
       software on", "motor on", "run chain on", "servo ...
       disconnect", "purged", "motor power on"]
59
60  # Liste med forskjellige symboler noder kan ha
```

```
61 shapes = ["box", "octagon"]
62
63 # Antall kjeder
64 num_chains = 10
65
66 xml_output = create_graph(num_chains, labels, shapes)
67
68 # Skriver til fil
69 file_name = "testgraph.xml"
70 with open(file_name, "w") as f:
71     f.write(xml_output)
```

Kode A.1: Python-skript for generering av konfigurasjonsfil.