NTNU
Det skapende universitet

Programvare for autonome arbeidende undervannsfartøy

Plattformuavhengig innlasting av dynamiske programvarebibliotek under kjøretid

Sigbjørn Aukland
Christian Anders Finnstrom

Master i ingeniørvitenskap og IKT
Innlevert: juni 2015
Hovedveileder: Sven Fjeldaas, IPM

Norges teknisk-naturvitenskapelige universitet
Institutt for produktutvikling og materialer

Forord

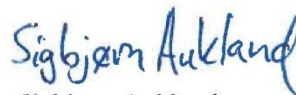
Ordet «bibliotek» vil ikke lenger først og fremst skape assosiasjoner til betonggulv, sittegrupper, stillhet og overveldende bokutvalg, men snarere med de – nå – nært forbundne ordene «statisk», «dynamisk» og «dinking».

Høsten 2014 begynte vi på en prosjektoppgave med mål om å utvikle et rammeverk for styring av servomotorer over Internett. Da vi var ferdige, føltes det litt som å sitte igjen med en arm uten kropp: Den hadde potensiale til å gripe, kjenne, og flytte seg rundt, men bare lå der. For å parafrasere et av vår veileder, professor Sven Fjeldaas' mange gode sitat: «Her ligger jeg og kan ikke annet». Vi ville få liv i armen.

GeoMOD er et program som kan brukes til å simulere, styre og planlegge roboters bevegelser. Vi skjønte raskt at dette programmet hadde potensiale for å kobles sammen med rammeverket vi hadde utviklet i prosjektoppgaven. Det var nok også en av vår veileders langsiktige tanker, da han introduserte masteroppgaven for oss, med Qt-prosjekttittel «entrance_05». Det viste seg derimot å være en del som gjenstod med programmet, før dette ville bli mulig. Når vi nå ser tilbake, tenker vi at arbeidet vi har gjort med å nærme oss en programversjon av GeoMOD hvor dette er mulig, gjerne har vært mer lærerikt enn det ville ha vært om vi skulle ha fortsatt å jobbe direkte med styringsrammeverket.

Det føles godt å levere fra seg «entrance_06» og tenke at neste prosjektbidragsytere kanskje kan kombinere våre to bidrag fra femteklasse, og få liv i armen.

Christian Finnstrøm



Sigbjørn Aukland

NTNU, Trondheim

10.06.2015

SAMMENDRAG

Det geometriske modelleringsprogrammet GeoMOD, tilbyr funksjonalitet for visualisering og styring av geometriske modeller. Programmet skal kunne knyttes opp mot roboters aktuatorer og brukes som styresystem. Undervannsfartøy (ROV) er en spesielt aktuell gruppe roboter i den sammenheng.

Det er et mål at fartøy som benytter GeoMOD som styringssystem skal kunne arbeide autonomt. Programmet må derfor være i stand til å laste inn programtillegg basert på øyeblikkelige behov. Dette kravet danner grunnlaget for en sentral problemstilling i vår masteroppgave: Plattformuavhengig innlasting av dynamiske programvarebibliotek under kjøretid.

For å løse dette problemet, har vi studert statiske og dynamiske programvarebibliotekers virkemåter i Linux og Windows. Besvarelsen drøfter byggeprosessen rundt disse bibliotektypene i de to operativsystemene, samt linking til statiske bibliotek og teknikker for å laste inn C++-baserte dynamiske bibliotek ved hjelp av C-funksjonene *dlopen* i Linux og *LoadLibrary* i Windows. Med dette som bakteppe, gikk vi videre med programvareutviklingsrammeverket Qts verktøy for plattformuavhengig innlasting av bibliotek: *QLibrary* samler plattformspesifikke C-funksjoner for dynamisk innlasting i én felles klasse, og tilbyr disse gjennom et sett metoder som automatisk tilpasses gjeldende plattform. Når biblioteket er lastet inn, kan *QLibrary* brukes til å nå bibliotekets symboler. *QPluginLoader* baserer seg på *QLibrary*, men tilbyr utviklere tilgang til et rotkomponent-objekt i det innlastede biblioteket, en egenskap som ble helt sentral i vår løsning.

Vårt arbeid er et bidrag i en årelang prosess. Å benytte fornuftige designmønstre og programvarearkitekturer er essensielt for effektiv og solid programvare. Arkitektonisk baserer programmet seg først og fremst på *Model-view-controller*, som separerer presentasjon av data fra selve dataene. Videre har vi benyttet *extensibility pattern* for å gjøre programmet tilbøyelig for programtillegg, samt *abstract factory pattern* i kombinasjon med *C++-templates* for å opprette objekter basert på kildekode fra de innlastede programtilleggene.

Prosjektets lange livsløp har resultert i en hel del arbeid knyttet til portering fra eldre Qt-versjoner. Store deler av den utdaterte funksjonaliteten har kunnet erstattes med nye funksjoner direkte, men en del funksjonalitet har måttet implementeres på ny. Dette gjelder blant annet design av nye brukergrensesnitt. Ved prosjektovertakelse fungerte programmet bare i Linux. Under hele prosessen har plattformuavhengighet vært et hovedmål, og programmet fungerer nå også i Windows-operativsystemet med en operativ filutforsker.

Abstract
The geometric modelling program, GeoMOD, offers visualization and control mechanisms for geometric models. The program can be used as a robotic actuator control system. Remotely operated underwater vehicles (ROVs) are a specially relevant group of robots in this context.

Robots utilizing GeoMOD as a control system, are supposed to operate autonomously. The program must therefore be able to load plug-ins as needed, based on continous demands. This requirement raises a central problem in our master thesis: Platform independent loading of dynamic libraries during runtime.

To solve this problem, we have studied static and dynamic software libraries, and how they function in Linux and Windows. Our thesis discusses the build process for these libraries, static linking, and techniques for loading C++ dynamic libraries through the C functions *dlopen* in Linux, and *LoadLibrary* in Windows. With this relevant background information, we pursued with the platform independent tools for library loading provided by the application framework Qt. *QLibrary* combines the platform specific C functions for dynamic loading in one class, and offers a corresponding set of methods that is matched against the current platform. When a library is loaded into the program, QLibrary can solve the library's symbols. *QPluginLoader* is based on *QLibrary*, but offers software developers access to a root-component object in the loaded library. This is an important feature, used in our application.

Our work is part of a long-lasting collaboration. Choosing sensible design patterns and software architectures is an essential part of developing effective and sustaining software. The program is based on the *Model-view-controller* architectural pattern, which separates data presentation from the actual data. We have also used the *extensibility pattern* in order to make the program compatible with extensions. Furthermore, we have used a combination of the *abstract factory pattern* and *C++-templates* to initialize objects based on source code from the loaded libraries.

Due to the long lifespan of this project, porting from old Qt versions has resulted in strenuous work. Most of the outdated functions, have been replaced by new versions. Other functions have required re-implementations. This has been the case for graphical user interfaces. When we received the project, the program was only functional in Linux. Platform independency has been our main goal throughout the process, and the program is now also running in Windows, with a functional filebrowser.

INNHOLDSFORTEGNELSE

Forord I

Sammendrag II

Abstract III

Introduksjon 6

Programvareutviklingsverktøy 7

- Programmeringsspråket C++ 7
- Programvareutviklingsrammeverket Qt 9
  - Qt-datatyper 10
  - Meta-Object System 10
  - Utviklerverktøyet Qt Creator 12

Designmønstre og programvarearkitektur 13

- SOLID – designprinsipper 13
- MVC – Model-view-controller 15
  - Qt Model-view 16
- Extensibility pattern 17
  - Plug-in pattern 17
- Abstract factory pattern 19
  - Template-konseptet i C++ 20

Programvarebibliotek 25

- Minnehåndtering – loader 25
- Statiske bibliotek 26
- Dynamiske bibliotek 27
  - Windows 28
  - Linux 29
- Statiske og dynamiske bibliotekers bruksområder 31

Lasting av programvarebibliotek 33

- James Nortons tilnærming til lasting av dynamiske bibliotek i Linux 33
  - Nortons innlastingsmetode i Windows 40

Programvarebibliotek i Qt 41

Dynamisk lasting av programtillegg i Qt.....41
QLibrary – den mest manuelle tilnærmingen i Qt.....41
QPluginLoader – samler bibliotekets symboler i et rotobjekt.....43
Hvordan opprette Qt Plugins.....48
Våre programvarebibliotek.....50
Collection.....51
Dynamiske testbibliotek.....51
Oppdatering av programvarebibliotek under kjøretid.....52
Pyramid_01.....53
Grafisk brukergrensesnitt.....54
Opprinnelig design.....55
Manager-vinduene.....55
Filutforsker-vinduet.....56
Info-vinduet.....57
Kompilering, linking og distribuering i Windows.....59
Innenfor Qt Creator.....59
Windows kommandolinje.....59
Qt-spesifikke DLLer.....60
C++-kompilatorspesifikke DLLer.....60
DLLer lastet av Qt selv.....60
Oppsummering og anbefalinger til videre arbeid.....62
Kommunikasjon mellom dynamiske bibliotek.....62
Kobling til Dynamixel-aktuatorer.....63
Collection.....64
Kilder.....65

INTRODUKSJON

GeoMOD er et geometrisk modelleringsprogram, hvis formål er å visualisere og styre en eller flere modeller, enten simulert, eller i sanntid. Modellene kan være vilkårlige geometriske modeller av lenkede mekanismer, som kan brukes til å representere virkelige roboter, deriblant undervannsfartøy.

Roboter som er utstyrt med slik programvare, skal kunne respondere på endringer i omgivelsene på en adekvat måte. Hvis et undervannsfartøy arbeider autonomt, vil det på bakgrunn av kontinuerlige vurderinger vite hva slags funksjonalitet det har behov for. Dette er den beste måten fartøyet kan ruste seg mot uventede utfordringer. GeoMOD baserer i stor grad sin funksjonalitet på innlastede programvarebibliotek; hovedprogrammet fungerer som et rammeverk som kan laste inn programvarebibliotek etter behov. Programmet tar utgangspunkt i tre typer programtillegg: Database, View og Tools. Database-plugins inneholder geometrisk informasjon om modeller. View-plugins brukes til å visualisere modeller, og kan presentere dem fra forskjellige synsvinkler. Tools definerer baner modellene og synspunkter kan bevege seg langs.

GeoMOD-prosjektet har pågått i mer enn ti år, noe som innebærer at både maskinvare og programvare har endret seg i løpet av dets levetid. Programutgaven vi overtok, var basert på Linux-operativsystemet. Rammen rundt oppgaven har først og fremst vært å portere programmet til Windows-operativsystemet, uten å ødelegge Linux-kompatibiliteten. Her kommer det plattformuavhengige utviklerverktøyet Qt inn i bildet: Da vi overtok prosjektet, bestod det av kildekoden som fungerte i Linux. Enkelte deler av denne fungerte også i Windows, takket være Qt, mens andre deler av programmet – især de som var knyttet til fillesing og lasting av programvarebibliotek, må kunne karakteriseres som å ha vært på «de evige jaktmarker». For å få disse programdelene reinkarnert i Windows, var vi avhengig av å sette oss inn i operativsystemenes ulike tilnærminger til programvarebibliotek, og i hva Qt-rammeverket kunne tilby på området.

Vi har tolket oppgaven vår dit hen – i samråd med veileder, professor Sven Fjeldaas – at de viktigste målene har vært å få programmet til å kjøre i Windows med en fungerende filleser som lar brukeren laste inn programtillegg, å oppbevare disse på en oversiktlig og hensiktsmessig måte i programmet, å oppdatere Qt-funksjonalitet til siste versjon (Qt 5,4), samt å redesigne de programdeler vi har funnet nødvendig – både programvarearkitektonisk, funksjonelt og grafisk.

PROGRAMVAREUTVIKLINGSVERKTØY

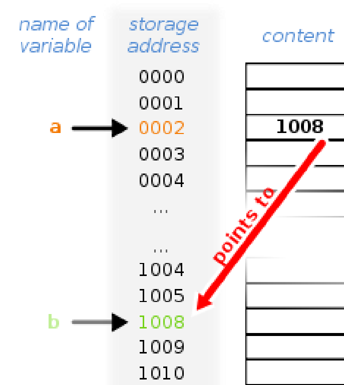
Ettersom prosjektet allerede var basert på programmeringsspråket C++ og programvareutviklingsrammeverket Qt, var det naturlig for oss å starte prosjektet med å sette oss inn i disse. Vi hadde begge erfaring med C++, men det er alltid utfordringer knyttet til å sette seg inn i andres kode. Qt hadde vi bare overflatisk kjennskap til, men med gode instruksjonsvideoer på YouTube, og stort sett god dokumentasjon på deres nettsider, lærte vi oss rammeverket å kjenne.

I delkapitlene som følger, introduserer vi disse verktøyene, og funksjonalitet som har vært særlig sentral for oss.

PROGRAMMERINGSSPRÅKET C++

Utviklingen av programmeringsspråket C++ startet med danske Bjarne Stroustrups doktorgradsarbeid i 1979. Da arbeidet han med Simula 67, et norskutviklet, objektorientert programmeringsspråk med simuleringer som hovedfokus. Bjarne Bjostrup vurderte det objektorienterte aspektet ved dette programmeringsspråket for å være nyttig også for generell programvareutvikling, men Simula hadde for dårlig ytelse til at det kunne brukes i praksis. Derfor startet han arbeidet med å lage en C-versjon med støtte for klasser, som i 1983 hadde resultert i programmeringsspråket C++. «++» er i C en operator for inkrement, dette antas å være Strostroups bakgrunn for navnevalget[1].

I likhet med C, støtter C++ pekere til dataverdier og metoder. En peker er et objekt med verdi som peker til en annen verdi lagret et vilkårlig sted i minnet. (Pointer (computer programming), http://en.wikipedia.org/wiki/Pointer_%28 computer_programming%29]. Som figur 1 viser, henviser peker **a** til en lokasjon i minnet knyttet til variabelen **b**. Pekere kan også peke til metoder, da kalles de *metodepekere*. Den siste typen pekere vi befatter oss med i prosjektet, er *opaque* pekere. Dette er pekere som peker til en datastruktur av uspesifisert type [3]. I kapittelet «Programvarebibliotek» går vi nærmere inn på hvordan pekere kan bli brukt for å aksessere metoder i dynamiske bibliotek.



Figur 1 – Pekerillustrasjon [2]

I tillegg til objektorientering og pekere, har vi også benyttet annen funksjonalitet som ble introdusert i C++, blant annet kan følgende nevnes [1]: *Function overloading*, det vil si at flere funksjoner kan ha samme navn, men forskjellige implementasjoner, hvor parametre avgjør hvilken versjon som skal brukes. Denne funksjonaliteten har en sentral rolle i vårt prosjekt ved utviklingen av *Abstract Factory Pattern*, som forklares i kapittelet «Programvarearkitektur». Referanser med ampersand-symbol (&), som lar brukeren deklarere flere variabler for samme data: Istedenfor å sende kopier av variabler som parameter til funksjoner, kan referanser til de opprinnelige dataene sendes, noe som reduserer både arbeidet med å lage kopier, og ikke minst minneforbruket. Dermed kan programmets ytelse øke. I vårt arbeid brukte vi ved noen anledninger referanser som parametre til metoder som traverserte gjennom lister. *const*-nøkkelordet ble også introdusert, som lar brukeren definere at data ikke skal kunne endres etter initialisering – de forblir konstante, i motsetning til variabler, som kan endres. Programdeler som er markert med *static*, eksisterer uavhengig av objekter – som noe for seg selv – slik at alle instanser og deler av programmet vil ha tilgang på den samme utgaven. I tillegg ble det mulig å *arve* fra flere klasser samtidig. I objektorientert programmering, er arv når en klasse overtar egenskapene til en annen klasse, som allerede er definert [4]. De arvede egenskapene kan spesialiseres i den nye klassen. Klasser kan i C++ deles i to hovedgrupper: *Konkrete klasser* og *abstrakte klasser*. I konkrete klasser er alle definerte metoder implementert. Det vil si at metodenes innhold er konkretisert. I abstrakte klasser, er én eller flere metoder definert, men ikke implementert. Disse metodene kalles *virtuelle*. Dersom *alle* metodene er virtuelle, kalles klassen *Pure Abstract Base Class* [5]. Et annet navn for Pure Abstract Base Class, som blir brukt i forbindelse med andre objektorienterte programmeringsspråk, og som nå også begynner å bli akseptert i C++-terminologi, er *interface*. Det er viktig å poengtere at det her ikke er snakk om grafiske brukergrensesnitt, men altså klasser bestående utelukkende av virtuelle metoder. Vi synes skillet mellom abstrakt og Pure Abstract Base Class er av så stor signifikans, at vi har valgt å benytte interface-begrepet – som i hvert fall rent språklig markerer forskjellen tydeligere. Kort oppsummert forholder vi oss altså til tre typer klasser: konkrete klasser (eller bare klasser), abstrakte klasser (virtuelle *og* konkrete metoder) og interface (også kalt pure abstract base class; *kun* virtuelle metoder). Når vi i oppgaven skriver at en klasse *implementerer* et interface, er det i C++-praksis det samme som å si at den arver en pure abstract base class. Vi tar i bruk arv i store deler av programmet, og kommer nærmere tilbake til dette i kapitlene «Designmønstre og programvarearkitektur» og i underkapittelet «Dynamiske Qt Plugins», som viser et eksempel på arv i en kodesnutt av en klasse kalt *MyPlugin*.

C++ brukes i dag i en rekke sammenhenger [6]. Systemprogrammering er et hovedområde, som følge av god ytelse og fleksibilitet. Det brukes mye i forbindelse med kontrollsystemer hvor det stilles høye krav til opererbarhet. Det sosiale nettverket Facebook, med over 1,44 milliarder brukere (31. mars, 2015) [7], er også utviklet i C++. I et intervju [8] hevder Stroustrup at programmeringsspråket han har skapt er med på å redusere klimautslipp, gjennom at Facebook trenger mindre datakraft, og dermed også et mindre strømforbruk for å drive en serverpark basert på C++, sammenlignet med andre programmeringsspråk, som følge av dets høye ytelse. En del av C++ sin popularitet må også anses å komme som følge av tradisjon. Når språket fungerer godt, og det utvikles mye programvare over lang tid, vil det bli desto mer omfattende å bytte til et annet. Utvikling av ny styringsprogramvare til for eksempelvis biler – BMW bruker C++ – vil i stor grad være basert på gammel kode.

Programvareutviklingsrammeverket Qt

Qt er et plattformuavhengig rammeverk som brukes til utvikling av programvare. Qt har sitt utspring fra et norsk selskap kalt Trolltech i 1991. Selskapet ble senere kjøpt opp av Nokia og solgt videre til Digia i 2011. Selskapet er i dag Oslo-basert [9]. Opprinnelig kunne Qt-programmer kun programmeres i C++, men introduserte senere støtte for andre programmeringsspråk – deriblant Java – gjennom tredjepartsutviklere. Qt blir hovedsakelig brukt i forbindelse med utvikling av applikasjoner med behov for grafikk, da rammeverket tilbyr et bredt spekter av verktøy for datapresentasjon. Deriblant tilbys grafiske brukergrensesnitt som emulerer stilene brukt i forskjellige operativsystem. Med jevne mellomrom kommer det nye Qt-versjoner, hvor funksjonalitet oppdateres eller fjernes. Ved utgivelsen av Qt 5 ble mye av funksjonaliteten tilbudt av Qt 3 gjort om på eller til og med fjernet. Mye av koden i GeoMOD var basert på utdatert Qt 3-funksjonalitet da vi overtok prosjektet, og ville derfor ikke kompilere. Som følge av dette, har mye tid gått med til portering til Qt 5. Heldigvis beskriver Qts dokumentasjon hvordan man kan gjøre dette for mange av problemene [10].

Qt-rammeverket blir delt opp i ulike moduler. Disse modulene inneholder klasser og verktøy for å utvide funksjonaliteten til programmeringsspråket man bruker. *Qt Essentials* er den mest sentrale samlingen av moduler, og inneholder de mest brukte utvidelsene for utvikling. *Qt Core* er den eneste av disse modulene som er obligatorisk å importere, og inneholder blant annet en rekke Qt-spesifikke-datatyper og *Meta-object system*. Vi vil se nærmere på disse utvidelsene i de neste delkapitlene.

QT-DATATYPER

Qt Core inneholder en omfattende samling klasser med implementasjoner av Qt-datatyper. Disse datatypene likner ofte på standard C++-datatyper, slik som *std::string*, som lagrer bokstaver eller tegn, men er implementert på andre måter med mål om forbedret ytelse og brukervennlighet. Et eksempel som er aktuelt å trekke frem, er *QString*, som erstatter nettopp *std::string*-datatypen. *std::string* i C++ lagrer informasjon om ett tegn som en gruppe med binærsifre (vanligvis 8) i bytes [11]. Datamaskinen kan på flere måter oversette bytes til bokstaver. Dette kalles tegnkoding, og *std::string* opererer uavhengig av tegnkoding. Det betyr at hvis vi kaller *std::string::length*, vil kallet returnere antall bytes i tekststrengen, ikke antall tegn. Qts *QString* bruker tegnkodingen Unicode, som er ansett som industristandarden innen tegnkoding. *QString*-datatypen har mange tilhørende metoder som øker brukervennligheten ved programvareutvikling, blant annet *QString::length* som returnerer antall tegn i tekststrengen, *ikke* antall bytes. Den inneholder også metoder for å iterere gjennom tegnene i strengen, strengemanipulasjon, samt metoder for å konvertere mellom ulike tegnkodinger.

Qts datatyper opererer godt med hverandre, men er ikke nødvendigvis direkte kompatible med standard C++-datatyper. Det vil for eksempel ikke være mulig å lagre strenger av typen *std::string* i en *QList* – en Qt-datatype som lagrer verdier i en liste og tilbyr rask tilgang, innsetting og fjerning av elementer. Det er likevel stort sett en smal sak å konvertere standard C++-datatyper til Qt-datatyper. I vårt arbeid har vi nesten utelukkende brukt Qts datatyper.

META-OBJECT SYSTEM

Meta-Object System er en del av Qt Core i Qt-rammeverket som muliggjør utvidelsen av funksjonalitet som normalt ikke er støttet av C++-kompilatorer. Meta-object system består av tre komponenter: *QObject*-klassen, *Q_OBJECT*-makro-instruksjonen og *Meta-Object Compiler*. [12]

QObject er klassen alle andre Qt-klasser arver fra, og kalles en *baseklasse* [13]. QObject er sentral i Qts såkalte *Object Model*. Denne modellen har som mål å strukturere klasser som arver fra QObject i *objekttrær*. Når man oppretter et objekt av typen QObject, og spesifiserer at dette objektet har et annet objekt av type QObject som forelder, vil foreldre-objektet automatisk oppdatere sin liste med barn-objekt. Dette spesifiseres ved å sende foreldreobjektet som parameter i kallet til konstruktøren til et barn-objekt. Foreldre-objektet tar dermed eierskap over barn-objektene sine, og ved sletting av foreldre-objektet vil også barn-objektene bli slettet. Dette er en verdifull egenskap vi har tatt i bruk i vår applikasjon, som forenkler prosessen ved sletting av flere objekter som henger sammen.

En av de mest brukte egenskapene i Qts Object Model er *signals* og *slots*, som introduserer støtte for kommunikasjon mellom objekter [14]. Konseptet med signals og slots er at når et objekt – må være/arve QObject! – forandrer sin tilstand, sendes et signal med informasjon videre til metoder, kalt slots. Denne funksjonaliteten gjør det blant annet enkelt å implementere *observer pattern*, hvor objekter lytter til andre objekter som sender signaler når tilstanden til objektet endrer seg. Vi har brukt signals og slots i alle klasser som inneholder *widgets*. Widgets er interaksjonselementer mellom menneske og maskin. [15]. Når brukeren interagerer med en widget, sendes et signal til en slot-metode som bestemmer hva som skal gjøres. Signals og slots brukes også i prosessen med sletting av objekttrær, beskrevet i avsnittet over.

En annen egenskap ved Qts Object Model som har betydning for vår applikasjon, er *dynamisk casting*. Dynamisk casting betyr å endre typen til et objekt i kjøretid, og vil blant annet bli gjennomgått med eksempler i kapittelet «Lasting av programvarebibliotek».

Q_OBJECT-makro-instruksjonen er nødvendig for alle klasser som implementerer signals and slots, Qts dynamiske casting og andre verktøy tilbudt av Qts Meta-Object System [16]. En makro-instruksjon er en spesiell kommando som må stå rett etter klassedeklarasjonen i *header*-filen. En header-fil inneholder deklarasjoner av klasser og metoder, samt spesifisering av *innkapsling* av disse. Innkapsling er en programmeringsteknikk hvor man spesifiserer hvilke deler av applikasjonen som skal ha tilgang til et objekt og dets symboler. Dette gjøres ved å benytte spesifikatorer som – blant flere andre – *public* og *private*. Objekter og symboler spesifisert med public, er tilgjengelige for alle. private spesifiserer at symboler kun kan nås internt i objektet. Q_OBJECT-makroen forteller *Meta-Object Compiler*, eller *MOC*, at klassen skal behandles som et Meta-Objekt System.

Meta-Object Compiler er en preprosessor; et program som prosesserer inndata og produserer utmatinger (eng.: output) som blir brukt som inndata i et annet program. [17]. Utmatingen sies å være den preprosseserte versjonen av inndataene, som videre blir brukt av påfølgende programmer, som andre kompilatorer. I motsetning til en tradisjonell kompilator, vil ikke MOC konvertere kildekode til maskinkode direkte, men til en ny C++-kildekodefil, som deretter kan mates inn i en C++-kompilator som produserer maskinkode[18]. MOC leser gjennom alle header-filer og leter etter klassedeklarasjoner som inneholder Q_OBJECT makro-instruksjonen. Når den støter på en slik fil, genererer MOC en C++ kildekode-fil som inneholder *Meta-object Code*. Med Meta-Object Code

menes C++-koden som ligger «bak kulissene» til funksjonaliteten støttet av Meta-Object System, som for eksempel signals og slots [19].

Detaljer rundt implementasjon av makro-instruksjoner knyttet til programtillegg-utvikling blir gjennomgått i kapitelene «Programvarebibliotek i Qt».

Utviklerverktøyet Qt Creator

Qt Creator er et *integrert utviklingsmiljø* (eng.: *IDE – integrated developer environment*) som lar deg skape Qt-applikasjoner[20]. Et integrert utviklingsverktøy er en type programvare som brukes til å skrive og lage applikasjoner, og inneholder typisk en teksteditor, kompilator og et feilsøkingsverktøy. Applikasjoner utviklet i Qt Creator kan være skrevet i programmeringsspråkene C++, JavaScript og QML i operativsystemene Linux, Windows og OS X. Qt Creator inneholder blant annet støtte for feilsøking, syntaks-hjelp og kodefullføring. Qt Creator bruker C++-kompilatoren fra GNU Compiler Collection (GCC) i Linux. I Windows er MinGW- eller MSVC – kompilatorer de mest brukte. I vårt arbeid har vi brukt MinGW-kompilatoren i Qt Creator.

MinGW-kompilatoren

MinGW, kort for *Minimalist GNU for Windows*, er en prosess og en samling verktøy for å lage et programvareprodukt. MinGW inneholder en GNU Compiler Collection- (GCC) kompilator og et sett fritt distribuerte Windows-spesifikke header-filer og statiske bibliotek, som muliggjør bruken av Windows API (se «Dynamiske bibliotek – Windows») [21]. GCC er en samling kompilatorer som støtter en rekke programmeringsspråk, deriblant C og C++. GCC ble opprinnelig utviklet for Linux, men porteringer til Windows tilbys også, blant annet gjennom MinGW, som vi benytter [22].

DESIGNMØNSTRE OG PROGRAMVAREARKITEKTUR

Innen programvareutvikling, er et *designmønster* en løsning på et tilbakevendende problem som opptrer i bestemte sammenhenger. Bruk av designmønstre kan lette programvareutviklingsprosesser, da de tilbyr – ofte enkle – mekanismer for å løse sammensatte problemer. I tillegg gjør de det mulig å beskrive et programs arkitektur nokså konsist, noe som er fordelaktig i prosjekter som løper over lengre tid og hvor bidragsytere kommer og går. [23]

De viktigste kriteriene for valg av programvarearkitekturer og designmønstre, er at programdelene skal ha avgrensede ansvarsområder, være oversiktlige og tilby konkrete løsninger på delproblemer. Vi anser det som bedre å tilby et omfangsrikt sett med klasser og metoder som løser delproblemer av større problem, enn å ha få klasser med tunge metoder som løser omfattende problemer selv, uten å bryte problemet opp i delproblemer som kan løses av andre, spesifikke metoder. Selv om det raskt blir mange klasser og metoder med denne tilnærmingen, vil man på sikt unngå at samme funksjonalitet blir implementert gjentatte ganger, og det vil ikke minst gjøre programvareutviklingen smidigere ved at man lettere kan oppdage eventuelle feil.

I arbeidet vårt har vi benyttet designmønstrene *MVC – Model-view-controller*, *Extensibility pattern*, *Plug-in pattern* og *Abstract factory pattern*, samt hatt fokus på designprinsippene av akronymet *SOLID*. Vi vil i de følgende delkapitlene gå inn på hver av disse, og hvordan vi konkret har brukt dem i vårt prosjekt.

SOLID – DESIGNPRINSIPPER

Innen objektorientert programmering er *SOLID* et akronym brukt for å beskrive fem hovedprinsipper innen design og struktur av programvaren. De fem forkortelsene står for: *single responsibility*, *open-closed substitution*, *Liskov substitution*, *interface segregation* og *dependency inversion*. Prinsippene har som formål å gjøre systemet lettere å utvikle over tid, samt å gjøre koden lett forståelig. For å oppnå fordelene ved *SOLID*, er det viktig at alle prinsippene er fulgt, da de baserer seg på hverandre.

Single responsibility-prinsippet handler om ansvarsavgrensning [24]. Hver klasse og metode skal ha sitt ansvarsområde. En endring i problembeskrivelsen vil dermed bare kreve at direkte relaterte klasser og metoder må oppdateres. Et eksempel for å illustrere dette, kan hentes fra det økonomiske dagliglivet. For at en person skal finne ut hvor mye hun har tjent i løpet av et år, må det trekkes fra

skatt og diverse posteringer fra bruttolønnen. Det vil være naturlig å separere disse trekkene fra hverandre: Først trekkes gjerne skatt fra bruttolønnen, deretter barnebidrag og så videre. Til slutt sitter man igjen med nettolønn. Det er altså enklere – og ikke minst langt mer oversiktlig – å forholde seg til flere enkelttrekk, enn om man skulle gjøre hele utregningen i et stort jafs.

Det andre prinsippet – *open/closed principle* – sier at systemet skal være «open for extension, but closed for modification» [25]. Med dette menes at man tillater utvidelse av systemet, *uten* å endre den eksisterende kildekoden. Fordelen med dette er at man bare trenger å feilsøke ny kode, da man vet at den eksisterende kildekoden fungerer. Dersom man ikke følger dette prinsippet, øker omfanget av feilsøking betraktelig, ettersom man har flere feilkilder enn nødvendig. La oss – med et obskurt eksempel – fortsette skatteeksempelet i forrige avsnitt med å si at det om to år innføres en ny avgift for personer som har færre enn to barn. Da ville det være bedre om man legger til dette som en *ny* postering, framfor å endre på de gamle. I det store skatteberegningsprogrammet legges det da simpelthen til en sjekk på hvorvidt personen skal påvirkes av posteringen eller ikke, ellers kan programmet gå i sin vante gang.

Liskov substitution-prinsippet sier at arv skal utvide funksjonalitet, ikke erstatte [26] . Et gjengivende eksempel hvor prinsippet er brutt, er en kvadrat-klasse som arver fra en rektangel-klasse. Kvadrat-klassen vil *erstatte* funksjonaliteten til rektangel-klassen ved å endre høyde- og bredde-variablene med side-variabel i kvadrat-klassen. Dette vil bryte med Liskovs substitusjonsprinsipp, fordi et kvadrat ikke har de samme attributtene som et rektangel.

Interface-segration-prinsippet tar sikte på at grensesnitt skal bli laget mest mulig spesifikke [27]. En bruker skal kun trenge å bry seg om de metodene som er aktuelle for henne, derfor kan det være en ide å lage flere små, spesialiserte grensesnitt, framfor store og altomfattende – dette oppfordrer interface-segration-prinsippet til. Som en presisjon, skal det sies at vi her ikke snakker om grensesnitt i grafisk sammenheng, såkalte *GUI* (graphical user interface/grafiske brukergrensesnitt), men grensesnitt som programvareutviklere forholder seg til.

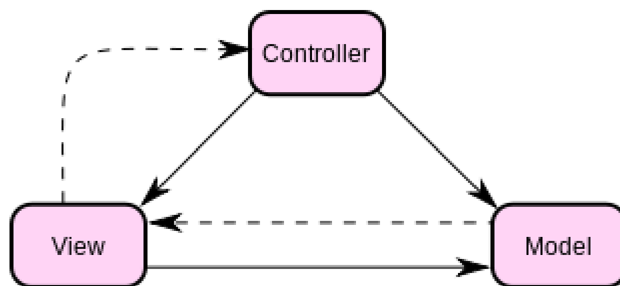
Dependency inversion-prinsippet oppfordrer til bruk av abstraksjoner: Høynivå-moduler bør ikke avhenge av lavnivå-moduler, men begge skal være avhengige av abstraksjoner [28]. Videre skal ikke abstraksjonene avhenge av detaljer; detaljer skal avhenge av abstraksjoner. Altså bør interaksjon mellom moduler på ulikt nivå foregå via abstraksjoner, og abstraksjonene skal være minst mulig detaljerte. Sagt på en forenklet måte, er formålet med *dependency inversion*-prinsippet at man bør bruke

abstraksjoner i størst mulig grad, slik at koden blir så generell som mulig, og dermed mer anvendbar i flere sammenhenger. Dersom høynivå-modulene er avhengige av spesialiserte lavnivå-moduler direkte, og ikke gjennom abstraksjoner, vil det være vanskelig å gjenbruke høynivå-modulene for å løse andre, liknende problemer.

Disse fem prinsippene danner et godt grunnlag for oversiktlig og fornuftig kode, på tvers av de ulike arkitektoniske designmønstrene vi har benyttet.

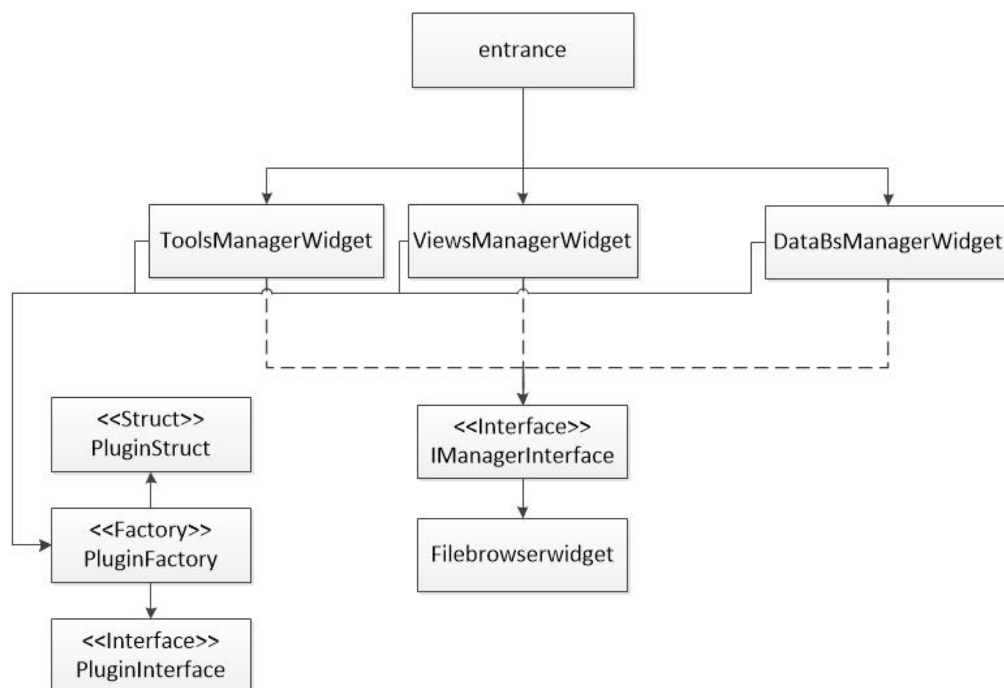
MVC – MODEL-VIEW-CONTROLLER

Målet med det arkitektoniske designmønsteret MVC, er å separere programdelene som går på arbeid på dataene (*model*), fra de delene av programmet som går på presentasjon av dataene (*view*). Den tredje delen, *controller*, tar seg av interaksjonen mellom model og view: Når brukeren klikker på en knapp på skjermen, henter controlleren aktuelle data fra model og gir dem til view, som viser dem fram, og om brukeren så gjør endringer på de presenterte dataene, vil controlleren gi beskjed til model om dette, slik at dataene oppdateres.



Figur 2 - Model-View-Controller arkitektur [29]

Med denne separasjonen vil endringer i brukergrensesnitt ikke føre til endringer i datahåndtering, og endringer i håndtering av dataene vil ikke endre presentasjonen av data. I vårt prosjekt har dette vært nyttig, da det har tillatt oss å arbeide med hver vår del av programmet samtidig, uten å måtte tenke så mye på hva slags endringer den andre gjør.



Figur 3 - Entrance_06 klassediagram

Av klassediagrammet fremkommer MVC-arkitekturen (figur 3). Som man ser, er dette den overordnede arkitekturen i programmet, hvor de tre *Manager*-klassene (*ToolsManagerWidget, ViewsManagerWidget, DataBsManagerWidget*) tilsvarer view-komponentene og PluginFactory tilsvarer model-komponenten i model-view-controller. De andre designmønstrene har blitt brukt for å løse mer spesifikke problemer ved programmet vårt.

QT MODEL-VIEW

Qt tilbyr en egen variant av MVC-arkitekturen, kalt *Qt Model/View* [38]. Her er controller-delen innebygget i view-delen, ved at såkalte «Qt view-widgets» har funksjonalitet som tolker inndata fra brukeren (f.eks. museklikk), og viderefører samsvarende kommandoer til model-delen. Der en MVC-arkitektur i konvensjonell betydning – i den grad man kan si at det finnes en konvensjonell betydning – gir en helhetlig separasjon mellom datastrukturer og brukergrensesnitt i programmet, fungerer Qts Model/View-arkitektur kun på Qt-datastrukturer. Altså: Den innebygde controlleren vil ikke fungere på vilkårlige datastrukturer [30]. I vårt program benytter vi egenkomponerte datastrukturer for oppbevaring av plugins; structer som består av navn (QString), filbane (QString) og plugin (se «Programvarebibliotek i Qt»). Dette gjør at vi må implementere interaksjonslogikken mellom model og view selv – controlleren – slik at vi ender med en mer «konvensjonell» MVC-arkitektur enn det Qt tilbyr.

EXTENSIBILITY PATTERN

I extensibility pattern har man et rammeverk (her: hovedprogrammet) som tilbyr løsning av et generelt problem, ved å knytte seg opp mot tilleggsmoduler som løser spesifikke delproblemer [31]. I vårt tilfelle vil det generelle problemet være modellering av roboter, mens eksempler på aktuelle delproblemer er kinematikk, geometri, Newtonsk mekanikk, styring av elektromotorer og visualisering.

Programmets kjerne ligger i *entrance*. Her er datastrukturer definert, så vel som en rekke metoder og klasser som blir benyttet både av hovedprogrammet og til interaksjon med programtillegg. Filleseren, som lar brukeren finne fram til et programtillegg i filsystemet på maskinen, og metoder for innlasting av disse under kjøretid, er eksempler på kjernefunksjonalitet man finner i hovedprogrammet.

Ytterligere funksjonalitet lastes inn i programmet som programtillegg, *extensions*. Disse programtilleggene kan inneholde klasser og metoder for presentasjon av geometriske modeller; eksempelvis robotarmer eller undervannsfartøy, posisjoneringsalgoritmer; ruteplanlegging og kinematikk, visualisering eller annet som man vil ha tilføyd kjerneprogrammet.

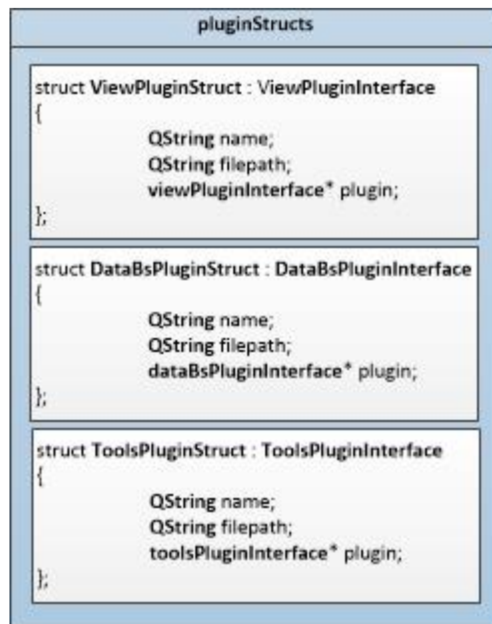
Et viktig moment med extensibility pattern er nettopp dette med tilføying av funksjonalitet: Nye metoder og klasser kan tilføyes programmet uten at det påvirker hovedprogrammets oppgave og virkemåte. I vårt tilfelle, hvor vi laster inn programtilleggene dynamisk under kjøring (se «Programvarebibliotek»), trenger ikke hovedprogrammet termineres selv ved store programoppdateringer.

PLUG-IN PATTERN

Programtillegg, eller plugins, er åpenbart sentrale komponenter i extensibility pattern. En ryddig måte å lage og bruke programtillegg på, er ved hjelp av designmønsteret *plug-in pattern* [32]. Her knyttes programtilleggene til hovedprogrammet via interface (klasser bestående utelukkende av virtuelle metoder). Programtilleggene implementerer aktuelle interface, og gir fullverdige versjoner av de virtuelle metodene. Hovedprogrammet inkluderer også disse interfacene, og vet av den grunn hvilke metoder som vil være tilgjengelige i programtilleggene.

Innlastede plugins oppbevares i structer bestående av navn, filbane og peker til programtillegg. Vi har i *pluginStructs.h* definert tre typer structer: *ViewPluginStruct, DataBsPluginStruct* og *ToolPluginStruct.*

Som man kan se (figur 4), er det typen programtillegg pekeren *plugin* peker til som er den eneste forskjellen i de tre structene:



Figur 4 – pluginStructs

Ettersom interfacene overlapper hverandre innholdsmessig, kunne vi valgt å bruke *ett* felles interface for alle programtilleggene, men vi vurderte separasjon å være lurt for å holde orden i den øvrige programstrukturen. Ved å ha tre interface – og følgelig tre structtyper – kan man forhindre at feil type programtillegg blir lastet inn på feil sted. For eksempel vil et forsøk på å laste inn et programtillegg av typen *ViewPluginInterface* i en *DatabasePluginStruct* stanses, og brukeren vil informeres.

Interfacene er i seg selv minimale, og metodene er så langt kun relatert til opprettelse og terminering av kontakt mellom hovedprogram og programtilleggene. Under følger ViewPluginInterface som et eksempel på dette.

```
class ViewPluginInterface
{
public:
    // returns a new instance, based on QPluginLoader root instance:
    virtual ViewPluginInterface* newInstance();
    // deletes _this_ instance:
    virtual void deleteInstance();
};
```

I Qt, må man legge til *Q_DECLARE_INTERFACE* (les mer om dette i «Programvarebibliotek i Qt - Hvordan opprette Qt Plugins»), slik at interfacet vil ende med å se slik ut:

```
class ViewPluginInterface
{
public:
    // returns a new instance, based on QPluginLoader root instance:
    virtual ViewPluginInterface* newInstance();
    // deletes _this_ instance:
    virtual void deleteInstance();
};

Q_DECLARE_INTERFACE(PluginInterface, "org.examples.PluginInterface.3")
```

De fleste kallene programtilleggene utfører, foregår enten internt, eller til det statiske biblioteket *Collection*, som inneholder metoder som mange av programtilleggene støtter seg på. I praksis innlemmes Collection-biblioteket i programtilleggene når de kompileres, sånn sett foregår også disse kallene internt. Ved senere utgaver av programmet, vil det gjerne bli nødvendig å muliggjøre interaksjon direkte mellom programtilleggene. Et tenkt scenario som vil kreve dette, er hvis to roboter skal bevege og posisjonere seg i forhold til hverandre. Men på det nåværende tidspunkt finner vi det tilstrekkelig med disse enkle interfacene som brukes i forbindelse med opprettelse og terminering.

ABSTRACT FACTORY PATTERN

Formålet med en *factory* er å opprette objekter av en konkret klasse. *Abstract factory pattern* tilbyr en måte å innkapsle en gruppe med individuelle factories som har et felles tema, uten å spesifisere deres konkrete klasser. [33]

I forrige delkapittel ble det forklart at vi har tre typer programtillegg, definert av interfacene *ViewPluginInterface*, *DataBsPluginInterface* og *ToolPluginInterface*. Istedenfor å lage en factory til hver av de tre programtilleggstypene, har vi laget en *abstract factory* kalt *PluginFactory*, som alle tre benytter. PluginFactory er en såkalt *template*-klasse, det innebærer at man kan definere klassens innhold og metoder uten å kjenne til hva slags konkret datatype den kommer til å operere på. Ettersom template-konseptet er sentralt i vårt program, går vi nærmere inn på det ved å vie det et underkapittel.

TEMPLATE-KONSEPTET I C++

Den planlagte factory-løsningen baserte seg på at vi skulle lage en abstrakt klasse bestående av både konkrete og virtuelle metoder for innlasting, instansiering og oppbevaring av programtillegg. Den abstrakte klassen skulle arves og implementeres av tre underklasser, en for hver type plugin struct; ViewPluginStruct, DataBsPluginStruct og ToolsPluginStruct. Mange av metodene i den planlagte abstrakte klassen viste seg å måtte arbeide direkte på structene av ulik type. Dermed måtte man altså deklarere tre varianter av hver metode, en for hver type struct. Vi vurderte da verdien av den abstrakte klassen å være minimal: Vi kunne like gjerne implementere alt i en felles klasse, eller fordele det på tre konkrete factory-klasser. Begge disse alternativene vurderte vi som dårlige, ettersom det ville være uoversiktlig å måtte sørge for å benytte korrekte metoder i fellesklassen. Samtidig ville innholdet i de tre klassene være nærmest identisk, den eneste forskjellen ville være hva slags struct-type de opererte på. Oppdateringer og vedlikehold ville måtte gjøres tre steder hver gang. Disse momentene tatt i betraktning, kombinert med at vi hadde dependency inversion-prinsippet (se «Designmønstre og programvarearkitektur»)i bakhodet, gjorde at vi konkluderte med at løsningen på en eller annen måte burde abstraheres. Dette ledet oss inn på template-konseptet i C++ [34].

I mange programmeringssammenhenger oppstår det et behov for at samme funksjonalitet skal utføres på forskjellige datatyper. Et enkelt eksempel er multiplikasjon. En multiplikasjonsmetode for heltall kan i C++ se ut som følger [35]:

```
int multiply1(int a, int b)
{
    return a * b;
}
```

En tilsvarende multiplikasjonsmetode for flyttall (her: double) kan se slik ut:

```
double multiply1(double a, double b)
{
    return a * b;
}
```

De to metodene har samme navn, og samme kropp, men arbeider på ulike typer data. I C++ er det lov å definere flere metoder med samme navn, så lenge parametrene er forskjellige. Dette kalles *overloading*. Hvilken metode som til syvende og sist skal kalles, avgjøres av hvilke parametre som

benyttes. De to kodelinjene i det følgende eksempelet kaller *multiyply1(a, b)* og skriver ut resultatet. Resultatet er gitt bak kommentartegnet.

```
std::cout << "Result: " << multiply1(5, 5); // Result: 25
std::cout << "Result: " << multiply1(5.01, 5.01); // Result: 25.1001
```

Merk automatikken: Kompilatoren tar «5» for å være en *int*, og kaller så opp den riktige varianten. «5.01» gjenkjennes som double, og riktig metode kalles.

Foreløpig har vi definert *multiply1*-metoden for heltall og flyttall. For en del metoder, kan det være et vilkårlig antall datatyper som benytter samme kropp. Istedenfor å lage implementasjoner for hver av disse, kan vi lage en felles implementasjon. Dette kan løses ved å bruke såkalte *templates* i C++.

Ved bruk av template-metoder, defineres ikke symbolers datatype før kompilering. Fordelen med dette er at man kan lage metoder som kan operere på flere typer data, så lenge syntaksen er lik. Dette er tilfellet for *multiply1*-funksjonene for heltall og flyttall, som ble presentert på forrige side. Disse kan slås sammen til én template-metode. [36]

Den generelle formen på en template-metode er som følger:

```
template <typename type> ret-type func-name(parameter list)
{
    // body of function
}
```

Her er *type* et plassholdernavn for den udefinerte datatypen metoden skal operere på. *typename* er det samme som en klasse: det kunne altså stått *class* istedenfor typename, men C++-komiteen valgte å innføre begrepet når man snakker om klasser i template-sammenheng [37]. *ret-type* er datatypen metoden skal returnere. Denne kan settes til *type*, om ønskelig. *func-name* er metodens navn, og *parameter list* erstattes med ønskede parametre, som også kan settes til *type*, om man ønsker å utsette konkretiseringen. Vi kan nå lage en ny, template-basert multiplikasjonsmetode som vil fungere på både heltall og flyttall:

```
template<typename type> type multiply2(type a, type b)
{
    return a * b;
}
```

Dersom vi gjør de samme kallene på *multiply2*-metoden, som vi gjorde på de overloadede *multiply1*-metodene, får vi følgende resultat:

```
std::cout << "Result: " << multiply2(5, 5); // Result: 25
std::cout << "Result: " << multiply2(5.01, 5.01); // Result: 25.1001
```

Altså gir den ut korrekte svar. Dette fungerer fordi kompilatoren ser hva slags datatyper som blir brukt ved kall til metoden, og lager instanser deretter. I vårt eksempel kalles metoden med *int* og *double*, dermed lager kompilatoren to instanser av funksjonen. Disse vil samsvare med de to *multiply1*-metodene. Dupliseringen, det å lage overload-metoder, overlates altså til kompilatoren. Vi ender altså med overloading i begge tilfellene, men template-konseptet lar oss – som programvareutviklere – forholde oss til én utgave av metoden. Dette forenkler vedlikeholdsarbeid betraktelig, og øker ikke minst kodens lesbarhet, da den blir mer oversiktlig enn om man har et hav av overload-metoder.

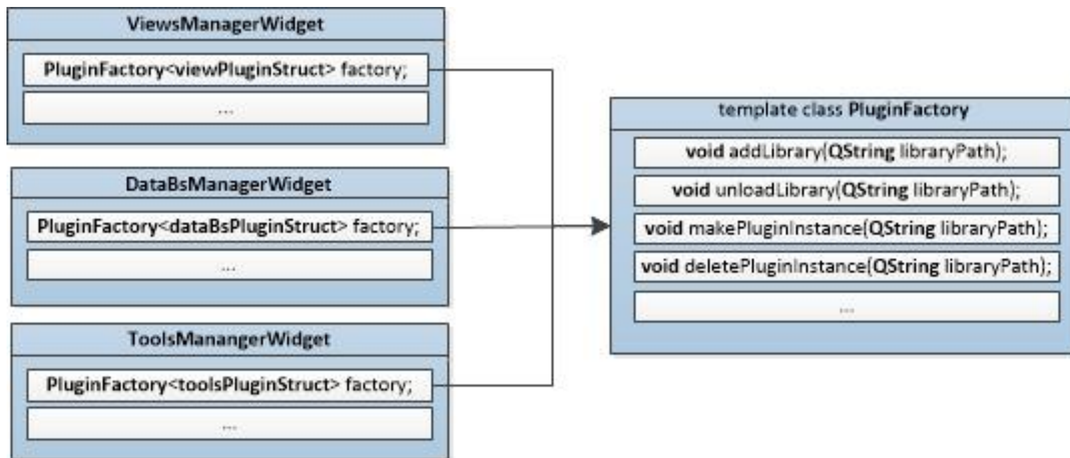
Hittil har template-metodene blitt brukt på enkle datatyper som heltall og flyttall. Templates kan også brukes i forbindelse med mer kompliserte metoder; det avgjørende er hvorvidt syntaksen – metodekroppen for å utføre den ønskede operasjonen – er den samme for hver datatype. Template-konseptet begrenses heller ikke til én template-type per metode, men kan utvides til et vilkårlig antall (… indikerer at man kan fortsette med flere template-typer, og er ikke del av syntaksen):

```
template <typename t1, typename t2, typename t3, ... >
```

I vårt prosjekt brukes templates ikke bare på enkeltmetoder, men også på hele klasser. Dette er tilfellet for *PluginFactory*. Da vil alle metoder i klassen kunne operere på template-typen(e) som er definert, man slipper altså å gjøre dette for hver metode. PluginFactory blir definert på følgende måte i *pluginfactory.h*:

```
template<typename pluginStructType> class PluginFactory
{
    ... // implemented code
}
```

Her er *pluginStructType* variabelen som ved kompilering fastsetter hva slags datatype instansen av PluginFactory skal arbeide på. Det opprettes en instans av PluginFactory for hver av de tre programtillegstypene; en i *ViewsManagerWidget* av typen *viewPluginStruct*, en i *DataBsManagerWidget* av typen *dataBsPluginStruct* og en i *ToolsManagerWidget* hvor typen er *toolPluginStruct* (se figur 5). Å bruke abstract factory fungerer fordi hver type (*viewPluginStruct*, *dataBsPluginStruct* og *toolPluginStruct*) har et felles sett med metoder som PluginFactory opererer på.



Figur 5 - Template og Struct arkitektur

Dette kan vises med *makePluginInstance*-metoden i *pluginfactory.h*, hvor *pluginStructType* ved kompilering blir omdannet til enten *View-*, *DataBs-* eller *ToolsPluginStruct*:

```
void makePluginInstance(QString filePath)
{
    pluginStructType newStruct(filePath,
                    generatePluginStructName_(filePath));
    instantiatePlugin_(newStruct);
    addPluginToPluginsList_(newStruct);
}
```

Her opprettes det en ny View-, DataBs- eller ToolsPluginStruct kalt *newStruct*. Dette gjøres ved å kalle structens konstruktør med filbane *(filePath)* til den aktuelle pluginen (programtillegg), og pluginstruct-navn som genereres av *generatePluginStructName_* basert på *filePath*. Deretter instansieres *newStruct* (se forklaring på *newInstance* i «Programvarebibliotek i Qt – QpluginLoader»), før den legges i listen med plugins.

Vi har utvidet factory-designmønsteret til å også tilby andre administrative funksjoner i forbindelse med programtillegg, deriblant funksjoner for å laste inn og ut programtillegg i minnet, containere for pluginstructene (View-, DataBs- og ToolPluginStruct) og tilhørende *set-* og *get*-metoder. I skatteberegningseksempelet som ble benyttet i starten av kapittelet, vil en set-funksjon kunne sette variabelverdier – som for eksempel bruttolønn, og en get-funksjon vil kunne brukes for å lese av hva verdien er.

Som illustrasjon 5 viser, ser man at hver *ManagerWidget* (View-, DataBs og ToolsManagerWidget) har sitt PluginFactory-objekt. Ved å benytte abstract factory pattern, har vi separert alt som har med lasting, oppbevaring, oppdatering og terminering av programtillegg fra ManagerWidget-klassene, slik

at metodene i disse kan være rettet mot deres ansvarsområde, som er å presentere dataene gjennom et brukergrensesnitt (ikke grafisk!) til programvareutvikleren.

PROGRAMVAREBIBLIOTEK

Et programvarebibliotek er en samling med programvare som kan bestå av metoder, variabler og klasser – heretter referert til under fellesbetegnelsen *symboler*, som ikke nødvendigvis utgjør et selvstendig program [39]. Programvarebibliotek kan brukes av annen programvare, for eksempel kan et matematikk-program ha nytte av å bruke programvarebibliotek med funksjoner for numerisk integrasjon eller trigonometri. Det som skiller kall gjort til et programvarebiblioteks metoder kontra interne metoder, er måten koden er organisert på i systemet.

Funksjonalitet i et programvarebibliotek er organisert slik at den kan brukes av flere programmer eller programdeler, som ikke nødvendigvis har noen øvrig tilknytning til hverandre [39]. Det vil si at flere programmer kan bruke de samme bibliotekene, noe som kan være gunstig ved programmeringsproblemer som går igjen. Gjenbruk av kode er udiskutabelt tidsbesparende i utviklingsfasen. En annen side ved dette er at man ikke trenger å forholde seg til selve kildekoden i bibliotekene, man trenger kun å vite hva slags funksjonalitet som tilbys.

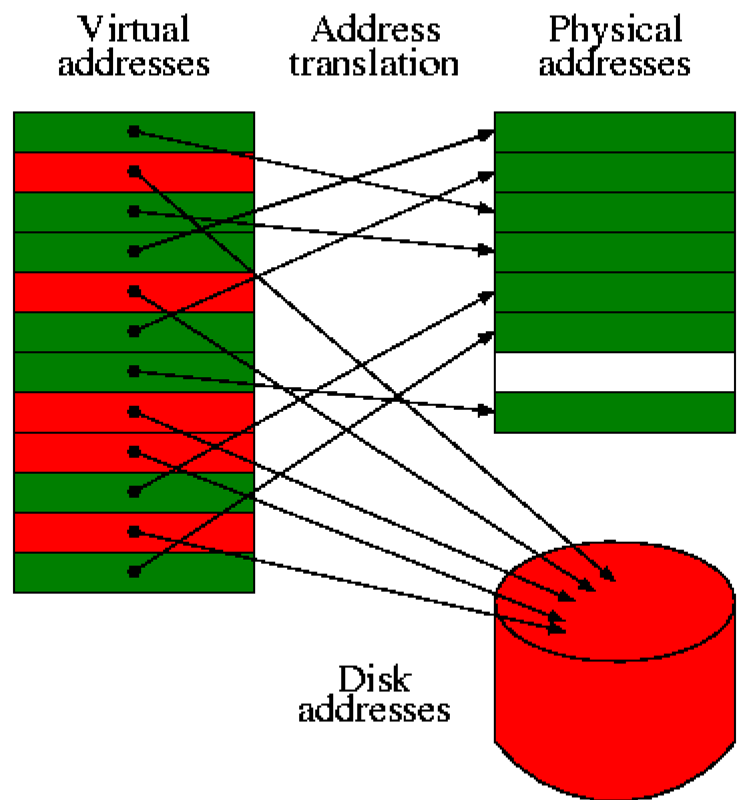
Et annet aspekt ved bruk av programvarebibliotek, og som står veldig sentralt i vårt prosjekt, er muligheten det gir for funksjonalitetsutvidelse. Programmet vi har arbeidet med bygger i stor grad på designmønsteret *extensibility pattern* (se kapittel «Designmønstre og programvarearkitektur – Extensibility Pattern»), som baserer seg på at et hovedprogram tilbyr en løsning på et generelt problem – i denne sammenheng geometrisk modellering av roboter – ved å benytte seg av programtillegg (programvarebibliotek) som står for løsningen av de spesifikke delproblemene. Eksempler på relaterte delproblemer er håndtering av hjørner, kanter, flater, posisjon og orientering.

I arbeidet med prosjekt- og masteroppgave har vi benyttet to typer programvarebibliotek, *statiske* og *dynamiske*. I de neste delkapitlene vil vi ta for oss disse, og gi en forklaring på deres styrker og svakheter, og ikke minst innvirkningen de har hatt på prosjektet vårt. Men først skal vi innom *loaderen*, som tar seg av minnehåndteringen i operativsystemet, og som er essensiell for at kjørende programmer skal kunne benytte programvarebibliotek.

MINNEHÅNDTERING – LOADER

For at kjørende programmer skal kunne bruke andre programdeler og bibliotek, må en såkalt *loader* sørge for at innholdet i de relevante filene blir tilgjengeliggjort i primærminnet. En loader er en del av operativsystemet som administrerer systemets minne. Behovet for loadere oppstår som følge av

at omfanget av programfiler og bibliotek overskrider primærminnets kapasitet. Dette løses ved å lagre filer i sekundærminnet, som har langt høyere lagringskapasitet enn primærminnet. Ulempen er at sekundærminnet har betydelig høyere aksesstid – tiden det tar fra data blir etterspurt til det er levert – og vil dermed gjøre programutførelsen urimelig mye tregere. Loaderen henter derfor inn ressurser fra sekundærminnet til det raske primærminnet når de behøves av andre prosesser, og frigir ressurser når de ikke lenger trengs. Noen bibliotek og programmer blir brukt så ofte, og av så mange prosesser, at de får permanente plasseringer i minnet. Disse kalles *memory-resident programs* [41]. For øvrige filer finnes det flere minnehåndteringsteknikker, deriblant *virtual memory*, som benyttes i både Microsoft Windows og Linux [42]. Kort fortalt går teknikken ut på å avbilde (eng.: map) virtuelle minneadresser til fysiske minneadresser. Programmene som kjører, forholder seg til de virtuelle adressene, og trenger ikke vite hvor dataene faktisk ligger i det fysiske minnet, det tar loaderen seg av. Som illustrasjon 6 viser, kan de virtuelle adressene peke til både primær- og sekundærminneadresser. Dette gjør det mulig for programmer å forestille seg at de har mer minne tilgjengelig enn de faktisk har. Når en prosess etterspør data fra en virtuell adresse, sørger loaderen for å «oversette» den virtuelle minneadressen til en fysisk adresse, og flytter om nødvendig dataene fra sekundærminnet inn i primærminnet [42].



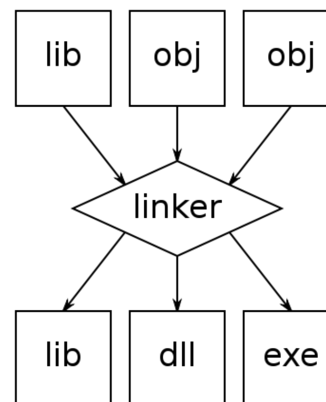
Figur 6 - Virtuelt minne [40]

STATISKE BIBLIOTEK

Et statisk bibliotek, eller *statisk linket bibliotek*, er et programvarebibliotek som blir innlemmet i et program når det *bygges* [43]. Med innlemming, menes at programmet og det statiske biblioteket flettes

sammen til et felles program. Denne prosessen kalles statisk bygging (eng.: *static build*) og består av *kompilering* og *linking*.

Et statisk bibliotek består av en eller flere *objektfiler*. Under kompilering transformeres kildekode skrevet i et programmeringsspråk til objektfiler av maskinkode, som en datamaskin kan forstå. [44] Objektfilene er ofte avhengige av andre objektfiler, og kan dermed være ubrukelige hver for seg. Derfor må man *linke* sammen kompilerte objektfiler til et kjørbart program [45]. Objektfilene består av tre typer symboler. Den første typen er *defined* eller *exportet*, det vil si at de er definerte i objektfilen, og tilgjengelige for andre moduler. Den andre typen symboler er *undefined* eller *imported*. Disse blir kalt eller referert til av objektfilen, uten at den kjenner til hvor og hvordan de er definert. Den tredje typen symboler er *local*; disse brukes internt i objektfilen, og er ikke tilgjengelig for andre. Første trinn i linkingen er å rydde opp i objektfilenes løse tråder; udefinerte symboler må adresseres. For å gjøre dette, henter linkeren objektfiler fra en samling kalt nettopp *programvarebibliotek*. Det endelige programmet innlemmer bare de objektfilene som trengs, ikke hele biblioteket [47].



Figur 7 – Linker [46]

Statiske bibliotek kan flettes sammen med andre bibliotek og objektfiler under linking for å danne et nytt bibliotek som en enkelt fil, eller de kan legges i reserverte minnelokasjoner – statisk minneallokering (memory resident-programs), minnelokasjonene bestemmes på tidspunktet ved kompilering/linking – og lastes av programmet under kjøring.

Dynamiske bibliotek

Hvordan linkingen gjøres er den største forskjellen på statiske og dynamiske programvarebibliotekers virkemåte. Der et statisk bibliotek linkes til et program i forbindelse med kompilering, laster en dynamisk linker (*dynamic linking loader*) inn et bibliotek til en kjørende prosess og binder dem sammen [48]. Dette kalles *late linking* eller *dynamisk linking*. Hvordan den dynamiske linkingen skjer i praksis, samt hvordan bibliotekene tilgjengeliggjøres i primærminnet (se loading i «Programvarebibliotek – Minnehåndtering – Loader»), avhenger av hvilket operativsystem systemet er basert på.

Windows

Følgende avsnitt er basert på Wikipedia-artikkelen «Dynamic-link library» [49] og Microsofts forklaringer på sine hjemmesider, i artikkelen «Dynamic-Link Library Creation» [50].

Microsofts implementasjon av dynamiske bibliotek kalles *Dynamic-Link Library* (DLL). I de første versjonene av Windows, hadde programmene delt adresseområde. Programmene skulle kunne samarbeide med hverandre ved å dele på prosessorens regnekraft. På denne måten kunne det grafiske brukergrensesnittet være responsivt for brukerinput. Det underliggende operativsystemet, *MS-DOS*, tok seg av systemoperasjonene, mens operasjoner på høyere nivå ble utført av Windows Libraries, «Dynamic Link Library». *API*et (Application Programming Interface) for presentasjon av grafikk, *GDI*, ble implementert i en DLL, og brukergrensesnittet i en annen DLL, *USER*. Disse DLLene måtte være tilgjengelige for alle programmene som kjørte, slik at de kunne samarbeide. Det ville for eksempel ikke være mulig for to ulike programmer å skrive grafikk til skjermen samtidig, dersom de ikke hadde felles tilgang til GDI. I tillegg ville det kreve for mye minne om de skulle ha sine egne kopier av bibliotekene. GDI måtte også transformere tegnekommandoer til operasjoner på forskjellige typer enheter. På skjermer skulle det tegnes bilder ved hjelp av piksler. Dersom noe skulle printes, trengtes andre kommandoer. Istedenfor å lage løsninger for alle typer enheter internt i GDI, så Microsoft det som en mulighet at GDI kunne laste inn spesifikke kodesnutter, tilpasset til de forskjellige enhetene, såkalte *enhetsdrivere*. Konseptet som ble brukt for å laste inn disse driverne, er det samme som ble gjeldende for lasting av andre programtillegg, og ble kalt nettopp *dynamisk linking*. Disse funksjonene – som vi kommer tilbake til i «Lasting av programvarebibliotek» – er i dag en del av det omfattende Windows APIet, definert i headerfilen *windows.h*, som utgjør kjernen i Windows-spesifikk programvareutvikling [51].

Som beskrevet, hadde programmene delt minneområde i de første versjonene av Windows. DLL-filer ble bare lastet inn i minnet én gang, og var deretter tilgjengelig for alle programmer. Ved å skrive og gjøre endringer i disse filene, kunne programmer samarbeide. Samtidig kunne et program ved uhell gjøre en fatal endring som ødela for de andre programmene. Ved innføring av 32- og senere 64-bits operativsystem, fikk hvert program sitt eget minneområde. Da kunne DLL-bibliotekene fortsatt bli delt, men dataene ble holdt private i hvert minneområde, med mindre DLL-filen eksplisitt krevde å dele også dataene (såkalte *shared sections*). Det ble altså opprettet lokale kopier av dataene for hver prosess, med mindre DLL-filen krevde deling.

Det finnes to hovedmåter å laste inn et dynamisk programtillegg i Windows [52]. Enten linkes biblioteket til når programmet kompileres (*implicit loading)*, eller så lastes det manuelt inn under kjøretid (*explicit loading*), og man får pekere til DLL-filens symboler. Når en DLL-fil linkes til ved bygging av programmet, krever en del kompilatorer et såkalt *import library*. En import-library-fil inneholder informasjon linkeren trenger for å avgjøre hvor de eksporterte DLL-filene ligger, slik at programmet kan lokalisere og bruke symbolene under kjøretid. Resultatet av linkingen mellom programmet og import-library-filen blir en fil som inneholder en *Import Address Table* (IAT), hvor all funksjonalitet i DLL-filen vil være henvist til. Når programmet så blir kjørt, oppdateres Import Address Table med gyldige adresser til symbolene i DLL-filen. På denne måten har man altså et slags statisk oppslagsverk som følger med DLL-filen. Man kan gjerne stusse over i hvor stor grad dette skiller seg fra et statisk bibliotek – særlig ettersom import-library-filer ofte har samme filendelse som statiske bibliotek – men forskjellen ligger i at det bare er adressene til symbolene i DLL-filen som innlemmes i programmet under linkingen, og ikke selve definisjonene, som er tilfellet for linking av statiske bibliotek.

Benytter man explicit loading, laster man inn DLL-filen under kjøring. Det trenger ikke å være noen som helst kontakt mellom programmet og DLL-filen(e) når programmet startes. Strengt tatt behøver ikke DLL-filen en gang å eksistere når programmet startes. «Linkingen» løses ved å bruke plattformspesifikke innlastingsmetoder (loadere, se «Programvarebibliotek – Minnehåndtering – Loader»), som gir pekere til de forskjellige symbolene i DLL-filen. Dette har vi i vårt prosjekt løst ved å bruke Qts plattformuavhengige klasser for innlasting av programtillegg. (se «Programvarebibliotek i Qt»).

LINUX

GeoMOD skal være et plattformuavhengig program, dermed skal den oppdaterte versjonen fortsette å fungere i Linux. Qt-rammeverket lar oss utvikle plattformuavhengig kode, og gjør mange av de plattformspesifikke tilpasningene for oss. Likevel har det å ha en god forståelse av hvordan Linux håndterer bruk av programvarebibliotek, vært av stor verdi.

I Linux kalles dynamiske programvarebibliotek *shared library*, og har filendelsen *so* (shared object) [53] [54]. Også her er det to hovedmetoder for innlasting av programtillegg: Enten lastes programtilleggene inn implisitt ved programstart, eller så lastes de inn eksplisitt under kjøring.

Bibliotek kan legges i standardmapper på Linux-operativsystemet, slik at de blir funnet og tilgjengeliggjort for all programvare. Ved å kjøre *ldconfig*-kommandoen, opprettes det en cache-fil (*/etc/ld.so.cache* → */etc/ld.so.conf*) som inneholder informasjon om hvordan de ulike bibliotekene kan nås. Ettersom det går raskt å lete gjennom en cache-fil, vil programmer på kort tid kunne ta i bruk so-filene. ldconfig-kommandoen må dog kjøres hver gang et nytt bibliotek legges til, slik at cache-filen oppdateres. Oppdatering av delte bibliotek kan gjøres i Linux uten å være bekymret for benyttende programmers kompatibilitet, fordi bibliotekversjonene nummereres. Dersom det blir gjort en endring som er så stor at biblioteket vil være inkompatibelt med den gamle versjonen, vil den nye versjonen automatisk få et nytt *versjonsnummer*. Dersom en mindre modifikasjon blir gjort, for eksempel en feiloppretting, vil det bli lagt til et såkalt *minor*-nummer i filnavnet. Når nye utgaver legges til, vil programmene som bruker biblioteket automatisk begynne å linke til siste versjon.

Man kan også benytte dynamiske bibliotek som ikke ligger på standardlokasjoner. I programmet vårt lastes de fleste programtillegg inn ved hjelp av en filleser, altså må fillokasjonene opplyses om i etterkant av programstart.

Å opprette en so-fil fungerer noenlunde på samme måte som for DLL-filer i Windows, men her trengs det ikke et import library, som er tilfellet for en del av kompilatorene i Windows. Først kompileres kildekoden til objektfiler med et kompileringsflagg som markerer koden som *posisjonsuavhengig* [55], det vil si at den skal fungere uavhengig av hvor i minnet den plasseres. Dette er en forutsetning for delte bibliotek. Deretter brukes et nytt flagg slik at objektkoden kompileres og linkes sammen til en shared library-fil.

Både Windows og Linux tilbyr altså delte, dynamiske bibliotek, men med to forskjellige tilnærminger, som følge av operativsystemenes ulike oppbygning. Målet er det samme: å tilby modularitet – at flere programmer og subprogrammer kan bruke den samme koden fra ett sted, istedenfor å måtte ha sine egne, innlemmede versjoner. Begge tilbyr implisitt og eksplisitt lasting av bibliotekene: Implisitt i Linux ved at man har oversikt over bibliotekenes plassering gjennom en cache-fil og i Windows gjennom programfilenes Import Address Table. Eksplisitt ved at man i begge operativsystemene kan bruke innlastingsmetoder som gir pekere til bibliotekenes symboler.

Utviklerverktøyet Qt skjuler mange av disse plattformspesifikke forskjellene for oss, og ordner opp «under panseret». Dette kommer vi nærmere tilbake til i kapittelet «Programvarebibliotek i Qt», hvor vi går nærmere inn på hvordan Qt kan brukes til å laste både statiske og dynamiske bibliotek.

Statiske og dynamiske bibliotekers bruksområder

Hver programdel som trenger symboler fra et statisk bibliotek, innlemmer en egen kopi av disse. En fordel med dette er robusthet; man vet at funksjonaliteten vil være tilgjengelig. Det er samtidig en ulempe fordi det tar mer plass i minnet med mange kopier, framfor å dele en versjon. En annen side ved bruk av statiske bibliotek, er at programdelene som benytter biblioteket må rekompileres dersom biblioteket oppdateres. På den ene siden kan dette medføre mye kompileringsarbeid, dersom det er mange brukere av biblioteket. På den annen side vil ikke feil i nye bibliotekversjoner automatisk påvirke, eller i verste fall ødelegge, annen programvare.

Fordelene og ulempene ved dynamiske bibliotek er i stor grad som ulempene og fordelene med statiske bibliotek – det er omstendighetene som avgjør om egenskapene er gode eller dårlige.

I vårt program benytter vi både statiske og dynamiske bibliotek. Programtilleggene som utvider GeoMODs funksjonalitet, lastes inn under kjøretid, og er laget som dynamiske bibliotek. Disse programtilleggene benytter et felles sett med klasser og metoder som vi har samlet i et bibliotek kalt *Collection*. Tanken var i første omgang at dette biblioteket skulle være dynamisk, da det brukes av mange programtillegg, og det på den måten kunne redusere behovet for minne. Det viste seg senere å oppstå et avgjørende problem i forbindelse med bruk av *template*-klasser (se «Oppsummering og anbefalinger til videre arbeid» for utfyllende beskrivelse av dette). Ved bruk av template-klasser, defineres ikke symbolers datatype før kompilering. Fordelen med dette er at man kan lage klasser og metoder som kan operere på flere typer data, istedenfor å ha utgaver for hver datatype. Det er likevel vesentlig at datatypene blir satt ved kompilering, ellers vil ikke systemet vite hvordan det skal fordele minneressurser. Ulike datatyper krever forskjellige mengder minne, og for å vite hvor mye plass som skal avsettes til, for eksempel, et array, må systemet vite hva arrayet skal inneholde. Ved bruk av statiske bibliotek, skjer, som sagt, fastsettelsen av symboler under linkingen, som er en del av byggeprosessen. Derfor er det uproblematisk å benytte template-klasser og -metoder i forbindelse med slike bibliotek. For dynamiske bibliotek, derimot, vil ikke bibliotekene vite av hverandres konkrete datatyper før under kjøring, og template-symbolene kan ikke konkretiseres. [56]. Dette resulterer i at tilbyderen av template-symboler (i vårt tilfelle Collection) ikke vet hvordan den skal håndtere kall av ulike datatyper: de konkrete utgavene eksisterer ikke. Vi har greid å finne det som kan se ut som løsninger på problemet, enkelte kompilatorer tilbyr omgåelser av problemet, men disse er gjerne kompliserte. Det gjelder blant annet MSVC-kompilatoren (Microsoft Visual C++)

[57]. Vi har ikke klart å finne løsninger med MinGW-kompilatoren som vi benytter i Qt, og har derfor valgt å gjøre Collection-biblioteket statisk.

Av fordelene og ulempene vi nevnte innledningsvis i dette kapittelet, er det særlig det at mange programtillegg bruker Collection som er prekært. Ved oppdateringer av statiske bibliotek, må alle programtilleggene rekompileres for å få nyeste utgave. Riktignok er det gjerne bare enkelte programtillegg som drar nytte av oppdateringene, og de må gjerne rekompileres uansett. Når det gjelder minnebesparelsene ved å bruke delte bibliotek, er det viktig å huske at hver programdel som benytter et statisk bibliotek, kun kopierer relevante symboler. Dermed blir ikke nødvendigvis overlappet veldig avgjørende.

Det at vi har fått til en løsning som fungerer ved å bruke statiske bibliotek, har ledet oss til den konklusjon at vi anser løsningen som den beste på det nåværende tidspunkt. Framtidige bidragsytere til prosjektet kan eventuelt vurdere verdien av å gjøre Collection-biblioteket dynamisk. Collection-biblioteket er en samling av *alle* symboler som brukes av de dynamiske bibliotekene, og bør uansett revideres, og også inndeles i underbibliotek. Dette har ikke vært en sentral del av vår oppgave, da vi har ansett det som mer fornuftig å i det hele tatt få programtilleggene opp og gå.

LASTING AV PROGRAMVAREBIBLIOTEK

Programtillegg utgjør det meste av GeoMODs funksjonalitet. Kjerneprogrammet brukes til å laste inn programtillegg for å løse spesifikke oppgaver som modellering og visualisering av roboter, kinematikk og posisjonering. Kapittelet «Programvarebibliotek» ga forklaringer på de ulike typene programvarebibliotek som er aktuelle i vårt prosjekt; statiske og dynamiske. Dette kapittelet handler om hvordan bibliotek kan benyttes i sammenheng med Qt-rammeverket; hvordan de lastes inn og ut av kjerneprogrammet, og hvordan man kan lage instanser av programvarebibliotek ved å benytte Qts plattformuavhengige verktøy.

Kapittelet «Programvarebibliotek i Qt – Hvordan opprette Qt Plugins» handler om hvordan man lager programtillegg som kan benyttes i Qt. Når man har laget programtillegg som Qt Plugins, tilbyr Qt noen hendige verktøy for plattformuavhengig innlasting, utlasting og symbol-løsning. Da vi overtok prosjektet, benyttet det ikke disse verktøyene. Det hadde heller ikke støtte for innlasting av dynamiske bibliotek i Windows. Vi starter derfor med en forklaring av hvordan mekanismen som ble benyttet på den gamle Linux-versjonen fungerer.

JAMES NORTONS TILNÆRMING TIL LASTING AV DYNAMISKE BIBLIOTEK I LINUX

Det følgende underkapittelet er i stor grad *basert* på professor James Nortons artikkel «Dynamic Class Loading for C++ on Linux», publisert 1. mai 2000 på Linux Journal [58].

Det eksisterer ikke mekanismer for lasting av programtillegg under kjøretid med C++. Derimot eksisterer plattformspesifikke mekanismer for dette i programmeringsspråket C. C er ikke et objektorientert språk, det vil – svært kort forklart – si at det ikke har mulighet til å lage strukturer (klasser) som samler attributter og metoder i objekter. C-kode kan stort sett brukes i C++, men ofte ikke omvendt. Fordelene med å bruke et objektorientert programmeringsspråk i programvareutvikling er i mange sammenhenger så store, at det oppstod et behov for å finne en løsning på hvordan man kunne bruke mekanismene for innlasting av programtillegg under kjøretid i C, i C++. På nettsiden «Linux Journal», presenterte professor James Norton den 1. mai, 2000, en elegant måte å gjøre nettopp dette på i Linux.

For C i Linux, eksisterer det en gruppe funksjoner for lasting av C-biblioteker under kjøretid, definert i *dlfcn.h* [59]. Denne headeren definerer følgende metoder for innlasting av dynamiske bibliotek [60]:

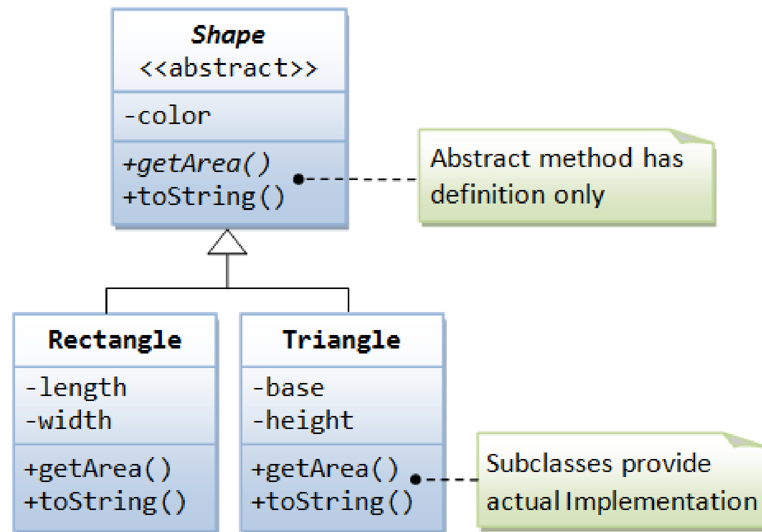
```
void *dlopen(const char *filename, int flag); // opens the file, flag decides
resolve option
void *dlsym(void *handle, char *symbol_to_find); // obtain pointer to function
(symbol)
char *dlerror(void); // if error: provides an error description
int dlclose(void *handle); // attempts to unload library if not used by others
```

dlopen-funksjonen laster inn et dynamisk bibliotek basert på *filename*, slik at symboler i filen kan bli aksessert ved bruk av *dlsym*-funksjonen. *flag* kan ha flere verdier, deriblant *RTLD_LAZY* og *RTLD_NOW*. Er *flag* satt til RTLD_LAZY, returnerer *dlopen* uten å prøve å løse symbolene i biblioteket. Dersom *flag* settes til RTLD_NOW, forsøker *dlopen* å løse udefinerte symboler i biblioteket, og returnerer en *opaque* peker til biblioteket, en såkalt *handle*. En opaque peker er en peker til en uspesifisert datatype (se «Programvareutviklingsverktøy – Programmeringsspråket C++»), og passer dermed her bra. Ved feil, returneres NULL. Da kan *dlerror*-funksjonen brukes for å hente ut informasjon om hva som gikk galt. *dlsym*-funksjonen brukes for å returnere en peker (også opaque) til et konkret symbol i biblioteket. Dette gjøres ved at funksjonen tar inn *handle* (fra *dlopen*), og forsøker å finne et symbol ved navn *symbol_to_find* i *handle*s adresseområde. Denne skal vise seg å bli helt sentral i Nortons løsning. *dlclose*-funksjonen dekrementerer en tellevariabel som holder rede på hvor mange som benytter biblioteket. Dersom tellevariabelen blir 0, lastes biblioteket ut – og *dlclose* returnerer 0. Hvis ikke biblioteket lastes ut, returneres et resultat ulikt null. Disse funksjonene danner grunnlaget for Nortons løsning.

I vår sammenheng, er det primært C++-bibliotek som skal lastes inn. Funksjonene som tilbys i C, kan også brukes i C++, men enkelte utfordringer oppstår. Man må sørge for at C-funksjonene finner fram til de korrekte symbolene i C++-biblioteket, ettersom måten symboler er organisert på, kan være forskjellig i de to programmeringsspråkene, som følge av C++' støtte for overloading (se «Programvareutviklingsverktøy – Programmeringsspråket C++»). Heldigvis finnes det en enkel løsning for dette: Ved å bruke kvalifikatoren *extern «C»* foran aktuelle metoder, vil linkingen foregå på C-vis, slik at C-metodene kan benyttes uten misforståelser (se «Programvarebibliotek i Qt – Qlibrary – den mest manuelle tilnærmingen i Qt»).

Et annet problem som oppstår, er at vi ikke kjenner til det vi ønsker å laste inn, i forkant av kompileringen: Hvordan skal man kunne lage objekter av noe man ikke vet hva er? Løsningen på dette er å bruke *interface*. Ved å lage et interface – i C++ kalles interface også *pure abstract base class* (se Programvareutviklingsverktøy – Programmeringsspråket C++») – bestående av virtuelle metoder,

som biblioteket implementerer (arver), kan programmet som laster inn biblioteket med sikkerhet vite at biblioteket *i hvert fall* tilbyr disse metodene. Et mye brukt eksempel på interface fremkommer av illustrasjonen til høyre. Her er *Shape* interfacet som implementeres i *Rectangle* og *Triangle*. Det at *Rectangle* og *Triangle* implementerer *Shape,* impliserer at disse må tilby fullverdige implementerte utgaver av de virtuelle metodene *Shape* har definert. I dette tilfellet metodene *getArea()* og *toString()*. La oss si at vi har en annen programdel, *ShapeContainer*, som har funksjonalitet for behandling og oppbevaring av diverse former *(Shape*-objekter), deriblant trekanter og rektangler. Istedenfor å spesifisere hva slags form formene har, trenger *ShapeContainer* strengt tatt ikke å vite mer enn at de er *Shape* (at de implementerer *Shape*-interfacet). Vi kan da for eksempel definere *ShapeContainer* slik:



Figur 8 - Eksempel på interface [61]

```
#include Shape.h // include interface, so that Shape is defined
class ShapeContainer
{
    std::list<Shape> shapes;

    void addShape(Shape toAdd)
    {
        list.push_back(toAdd);
    }
    // ...
}
```

Her kan *Shape* være *Rectangle*, *Triangle* eller en hvilken som helst annen form som implementerer *Shape*. *ShapeContainer* kjenner til hva en *Shape* er ved å inkludere *Shape.h*-filen, og kan dermed lagre nye *Shape*-objekter under kjøring, uten å vite hva deres faktiske implementasjoner er. Nettopp dette prinsippet er løsningen på hvordan man kan lage objekter ut av noe man ikke vet konkret hva er; i vårt tilfelle objekter knyttet til symboler i dynamiske bibliotek.

I Nortons teknikk, opprettes ikke disse objektene direkte i hovedprogrammet, men i bibliotekene selv. Ettersom både C++ og C har blitt nevnt i dette kapittelet, opplyser vi om at det her er C++-bibliotek som er gjeldende. Vi fortsetter å basere eksemplene på *Shape*-interfacet, men legger til en ny factory-metode (se «Designmønstre og programvarearkitektur – Abstract factory pattern») som kan lage nye objekter:

```
class Shape
{
    virtual Shape* maker()=0;
    //... other methods
}
```

maker-metoden vil, når den er implementert i en underklasse, returnere en peker til det konstruerte objektet. I *Rectangle* vil *maker*-metoden returnere en peker til et *Rectangle*-objekt, og implementeres på følgende måte:

```
class Rectangle : Shape // implements Shape
{
    Shape* maker()
    {
        return new Rectangle;
    }
    // ... other methods
}
```

Og for ordens skyld tar vi også med implementasjonen av *maker*-metoden i *Triangle*, som returnerer en peker til et *Triangle*-objekt:

```
class Triangle : Shape // implements Shape
{
    Shape* maker()
    {
        return new Triangle;
    }
    // ... other methods
}
```

Neste problemstilling er knyttet til hvordan vi kan få kontakt med *maker*-metodene, når de er i dynamiske bibliotek. Her kommer C-funksjonene for Linux forklart tidligere i kapittelet inn i bildet:

```
void *dlopen(const char *filename, int flag); // opens the file, flag decides
resolve option
void *dlsym(void *handle, char *symbol_to_find); // obtain pointer to function
(symbol)
char *dlerror(void); // if error: provides an error description
int dlclose(void *handle); // attempts to unload library if not used by others
```

Ved å bruke *dlopen* kan man sette *handle* som en opaque peker til et bibliotek, her *libThatILoad.so*. Deretter kan vi bruke *dlsym* med *handle* for å få en opaque peker til *maker*-funksjonen i biblioteket:

```
void *handle = dlopen("libThatILoad.so", RTLD_NOW); // attempt to get
opaque pointer to lib
if(handle == NULL) // if no handle was acquired
{
    cerr << dlerror() << endl; // write error message
    exit(-1);
}
void *mkr = dlsym(handle, "maker"); // obtain pointer to maker-function
```

Merk at *mkr* her er av typen **void*, og ikke *Shape*. Det er fordi pekeren er en opaque peker til funksjonen *maker* i *libThatILoad*, og ikke til et nytt *Shape*-objekt. Dersom vi nå vil ha tak i et *Shape*-objekt, kan vi benytte *mkr*-funksjonen på følgende måte:

```
Shape* my_shape = static_cast<Shape* ()>(mkr)();
```

Her kalles først *mkr*-funksjonen. Resultatet *castes* til typen *Shape*, og tilegnes til *my_shape*-variabelen. Casting vil si å endre typen til et objekt. *my_shape* er en *Shape*, men hva slags *Shape* den konkret er, avhenger av hvordan *maker*-funksjonen er implementert i *libThatILoad*. Den kan for eksempel være implementert til å returnere *Rectangle* eller *Triangle*.

Så langt har vi oppnådd kontakt med *maker*-funksjonene, slik at vi kan lage *Shape*-objekter. Prosedyren må sies å være litt knotete, særlig sammenlignet med hvordan man vanligvis instansierer objekter i C++. De neste trinnene gjør instansieringsprosessen betraktelig enklere, men teknikken kan ved første øyekast virke noe intrikat.

C++ tilbyr et verktøy kalt *map*. Map er containere som knytter en *key value* og en *mapped value* [62]. Key value brukes vanligvis som identifikator, mens mapped value er de aktuelle dataene. Et fint eksempel på map følger under:

```
map<string, int> telefonkatalog; // create a map < «name», «phone number » >

telefonkatalog["Jon Gelius"] = 99887766; // add a new pair in the map
```

Her er *string* key value (navn), og *int* er mapped value (telefonnummer).

I vår sammenheng, kan vi lage et globalt – tilgjengelig for alle relaterte programdeler – map bestående av klassenavn og peker til den klassens *maker*-funksjon, her kalt **maker_pointer()*. Vi kaller kombinasjonen klassenavn og maker_pointer() for et *par*. Vi velger å typedefinere (*typedef*)

*maker_pointer() til *Shape*, for å øke lesbarheten i koden. Dette gjør vi på følgende måte et vilkårlig sted i hovedprogrammet:

```
typedef Shape *maker_pointer(); // *maker_pointer() is now equivalent with Shape
map<string, maker_pointer> makerMap; // string is key, maker_pointer is value
```

Ved å legge inn par for de ulike formene vi har i *makerMap*, kan vi opprette *Shape*-objekter ved å benytte C++-map. La oss si at vi ønsker å lage et *Rectangle*-objekt:

```
Shape *newRectangle = makerMap["Rectangle"]; // Rectangle is key, *newRectangle is
value
```

Denne tilnærmingen lar oss organisere de ulike *maker*-funksjonene på en ryddig og tilgjengelig måte. Likevel kan det bli en del arbeid å legge alle parene inn i *makerMap*, spesielt dersom man har mange former. Kanskje vet man ikke en gang eksakt hvor mange former man har. Derfor kan man automatisere denne prosessen, slik at *maker*-funksjonene selv registrerer seg i *makerMap*. Måten dette ble gjort på i GeoMOD da vi overtok prosjektet, var ved å benytte egne proxy-klasser i hvert enkelt bibliotek, hvis eneste ansvar var å registrere bibliotekets *maker*-funksjon i *makerMap*:

```
class proxy
{
public:
    proxy(){ // class constructor
        makerMap["shape name"] = maker; // registers new pair in makerMap
    }
};
```

Som man ser av kildekoden, gjøres registreringen av det nye klassenavn-maker_pointer-paret i konstruktøren til proxy. Dette fungerer ettersom *makerMap* er globalt – det vil være tilgjengelig for alle tilknyttede programdeler. Ved å deklarere en instans av *proxy*-klassen, *proxy p*, i biblioteket, vil denne konstruktøren bli kalt når *dlopen*-funksjonen blir kalt med *RTLD_NOW*-flagget, som forsøker å løse alle symbolene i biblioteket. Et større utdrag av *libThatILoad*-biblioteket kan dermed se slik ut:

```
// This is the header and source code in libThatILoad-library

// Header:
#ifndef __rectangle.h
#define __rectangle.h
#include "shape.h"
class Rectangle : Shape // implements Shape
{
public:
    Shape *maker();
    // ... other methods
};
#endif // __rectangle.h

// Source code:
#include <iostream>
#include "rectangle.h"
extern "C" {
Shape *maker(){
    return new Rectangle;
}

// ... other methods

class proxy {
    public:
        proxy(){
            makerMap["Rectangle"] = maker; // register the maker in makerMap
    }
};

proxy p; // our proxy-instance(calls proxy constructor, which registers the maker
in makerMap)
}
```

Når et bibliotek lastes inn i det kjørende programmet med *dlopen*, vil *maker*-funksjonen automatisk bli lagt til i *makerMap*. Etter at biblioteket *har blitt lastet inn*, blir det dermed en smal sak å opprette objekter:

```
// somewhere in the main program:
Shape* my_triangle = makerMap["Rectangle"];
```

Når *my_triangle*-objektet så er opprettet, kan dets symboler nås på standard vis. La oss si at *Triangle* har en funksjon, *calculateArea*, som beregner trekantens areal. Da kan dette hentes ut slik:

```
std::cout << "The area of the triangle is: " << my_triangle->area();
```

Nortons tilnærming til lasting av dynamiske bibliotek automatiserer prosessen med å løse bibliotekenes symboler. Ved å bruke interface, blir det mulig å samle disse symbolene i objekter, til tross for at C – som er bibliotekinnlastingsmetodenes språk – ikke har støtte for dette. Ved til slutt å

samle pekerne til *maker*-funksjoner, hvis formål er å opprette objekter i bibliotekenes klasser, i C++-map (se «Programvarebibliotek – Minnehåndtering – Loader») som par med tilhørende klassenavn, må Nortons tilnærming sies å være en smart, logisk og elegant løsning på et komplekst problem.

NORTONS INNLASTINGSMETODE I WINDOWS

Ettersom GeoMOD skal være plattformuavhengig, har det vært behov for å få til en løsning som også fungerer på Windows-operativsystemet. Et kjerneproblem her, er at *dlopen*-funksjonene (definert i *dlfcn.h* (se forrige delkapittel) ikke er tilgjengelig i Windows. Derimot finnes det alternativer, som gjør samme jobben. Disse funksjonene stammer fra Windows-APIet, som er definert av *windows.h* (se «Programvarebibliotek – Dynamiske bibliotek - Windows»). Følgende oversikt, hentet fra Wikipedia-artikkelen «Dynamic loading» [51], viser disse:

Name	Standard POSIX/UNIX API	Microsoft Windows API
Header file inclusion	`#include <dlfcn.h>`	`#include <windows.h>`
Definitions for header	`dl` (`libdl.so`, `libdl.dylib`, etc. depending on the OS)	`kernel32.dll`
Loading the library	`dlopen`	`LoadLibrary` `LoadLibraryEx`
Extracting contents	`dlsym`	`GetProcAddress`
Unloading the library	`dlclose`	`FreeLibrary`

Figur 9 - Innlastingsfunksjoner i C

Prinsipielt kan altså Nortons metode også anvendes i Windows. Men istedenfor å supplere med en ny, Windows-basert versjon, valgte vi etter nøye vurdering å heller vie fokuset på å undersøke Qts plattformuavhengige verktøy for innlasting av programvarebibliotek.

PROGRAMVAREBIBLIOTEK I QT

Qt tilbyr funksjonalitet for plattformuavhengig import og eksport av dynamiske og statiske bibliotek. Forrige kapittel adresserer en del problemstillinger knyttet til dette temaet, særlig knyttet til C++' mangel på funksjonalitet for innlasting av dynamiske bibliotek, og i forbindelse med instansiering av objekter definert i bibliotekene. Det at Qt velger å lage dedikerte verktøy for dette formålet, viser at dette er et velkjent problem i programvareutvikling.

Vi skal i dette kapittelet se nærmere på disse verktøyene, og går inn på innlasting av dynamiske og statiske bibliotek, samt til slutt hvordan man kan opprette Qt-spesifikke statiske og dynamiske bibliotek, såkalte *Qt Plugins*.

DYNAMISK LASTING AV PROGRAMTILLEGG I QT

En fordel med å bruke rammeverk, er at de gjerne tilbyr fullgode verktøy som kan anvendes på kjente problemer som gjerne er omfattende å implementere manuelt. Samtidig er dokumentasjonen ofte begrenset til å belyse verktøyenes praktiske anvendelser, og ikke hvordan de *egentlig* virker. I mange sammenhenger er dette helt greit; fungerer det, så fungerer det. Men i forbindelse med en masteroppgave, er det av stor verdi å vite hva som skjer under panseret på verktøyene med de flotte navnene. Vi har forsøkt så godt vi kan, ved å kombinere Qts offisielle dokumentasjon, relaterte innlegg på diskusjonsforum og egne erfaringer, å komme litt under huden på Qts bibliotekinnlastingsfunksjoner.

QLIBRARY – DEN MEST MANUELLE TILNÆRMINGEN I QT

QLibrary er en Qt-klasse med metoder for plattformuavhengig innlasting av dynamiske bibliotek under kjøring [63]. *Én* instans av QLibrary knyttes til *ett* dynamisk bibliotek, som kan være av typen *so* for Linux (se «Programvarebibliotek – Dynamiske bibliotek – Linux»), *dll* (se «Programvarebibliotek – Dynamiske bibliotek – Windows») for Windows eller *dylib* for Mac. Bibliotekets type trenger ikke defineres, Qt vil se hva som er tilgjengelig, og tilpasse seg deretter. Dette gjør at man kan benytte eksakt samme kildekode på flere plattformer. Tilknytningen til et bibliotek kan gjøres på følgende måte ved initialisering av QLibrary-objekter:

```
QLibrary* myLib("libThatILoad"); // initialization
```

eller på følgende måte, dersom *myLib* er initialisert, men ikke knyttet til et bibliotek:

```
QLibrary* myLib; // initialization

// ...

myLib->setFileName("libThatILoad");
```

Flere instanser av QLibrary kan være knyttet til samme bibliotek.

Gangen i det hele likner mye på den manuelle tilnærmingen vi viste gjennom Nortons metode. Når et bibliotek er lastet inn, må bibliotekets symboler løses for at de skal kunne brukes av hovedprogrammet. Dette gjøres ved å benytte *resolve*-funksjonen [64]. Denne funksjonen returnerer – i likhet med *dlsym* (se «Lasting av programvarebibliotek – James Nortons tilnærming til lasting av dynamiske bibliotek i Linux») – adressen til symbolet den forsøker å løse, og 0 ved feil. I følgende eksempel forsøkes funksjonen *average* å løses og tilegnes til *avg*:

```
typedef int (*AverageFunction)(int, int); // (*AverageFunction)(int, int) ⇒ int

AverageFunction avg = (AverageFunction) myLib->resolve("average"); // resolve sym-
bol
if(avg) // if avg was succesfully resolved
    return avg(11, 9);
else
    return -1;
```

Først defineres det at *(*AverageFunction)(int, int)* er ekvivalent med *int*. Når man da skriver *AverageFunction avg*, vet man at *average* er en funksjon som skal returnere en *int*. Det første som skjer i *resolve*-kallet er at *myLib* kaller sin *resolve*-funksjon. Ettersom *myLib* er tilknyttet biblioteket *libThatILoad*, forsøker den å finne en peker til et symbol i dette biblioteket som samsvarer med navnet «average». Resultatet castes til typen *AverageFunction*, og tilegnes til *avg*. *resolve* returnerer 0 dersom symbolet ikke kunne løses.

Qts dokumentasjon opplyser at symboler i C++-bibliotek må merkes med kvalifikatoren *extern «C»* [65]. I Windows må symboler i tillegg merkes med *__declspec(dllexport)* for at de skal bli tilgjengelige for hovedprogrammet. Denne forskjellen trenger likevel ikke å resultere i et behov for to separate utgaver av kildekoden. Ved å bruke en makro som sjekker hvilken plattform som benyttes, kan samme kildekode brukes:

```
extern "C" MY_EXPORT int average(int a, int b)
{
    return (a + b) / 2;
}
```

Her defineres *MY_EXPORT* avhengig av plattform:

```
#ifdef Q_OS_WIN // if Windows
#define MY_EXPORT __declspec(dllexport)
#else // if other operating system
#define MY_EXPORT
#endif
```

Den siste metoden som i denne sammenheng er verdt å nevne, er *unload* [66]. Denne metoden brukes for å laste et bibliotek ut av programmet. Ettersom det kan eksistere flere QLibrary-instanser med tilknytning til samme bibliotek, vil kallet kun være vellykket (returnere *true*) når *unload* kalles på den siste gjenværende instansen. Dette samsvarer med virkemåten til *dlclose*-funksjonen (se «James Nortons tilnærming til lasting av dynamiske bibliotek i Linux») som ble brukt i Nortons metode.

I det hele tatt virker QLibrary i stor grad å være basert på *dlfcn.h*-funksjonene (se «James Nortons tilnærming til lasting av dynamiske bibliotek i Linux») og Windows API-funksjonene (definert i *windows.h*) for innlasting av dynamiske bibliotek i henholdsvis Linux og Windows. Det kan virke som om QLibrary fungerer som en slags *wrapper*-klasse for disse to (tilsvarende Mac-funksjoner kommer i tillegg), det vil si at den tilbyr funksjonaliteten i disse klassene samlet i én, som presenterer metodene på en generell måte, uavhengig av operativsystem. Altså: QLibrary sjekker hva slags plattform programmet kjøres på, og velger hvilke metoder som skal brukes deretter. Denne påstanden kan ytterliggere argumenteres for ved å for eksempel se på *QLibrary::errorString*-metoden, som på samme måte som *error*-funksjonen definert i *dlfcn.h* returnerer en feilmelding.

En helhetlig løsning kunne vært å kombinere QLibrarys plattformuavhengige innlastingsmetoder med Nortons teknikk for å samle bibliotekers symboler i objekter. Men her tilbyr Qt et annet verktøy, kalt *QPluginLoader*, som gjør dette for oss.

QPLUGINLOADER – SAMLER BIBLIOTEKETS SYMBOLER I ET ROTOBJEKT

QPluginLoader har samme hovedoppgave som QLibrary – å laste inn programtillegg under kjøring, plattformuavhengig [67]. Én instans av QPluginLoader kan bare operere på ett bibliotek om gangen. Men man kan ha flere instanser av QPluginLoader som hver jobber på sitt bibliotek, eller man kan ha flere instanser som jobber på samme bibliotek. En QPluginLoader-instans initialiseres på følgende måte:

```
QPluginLoader* pluginLoader("libThatILoad"); // initialization
```

eller det kan – i likhet med QLibrary – initialiseres uten å knyttes til en fil med en gang:

```
QPluginLoader* pluginLoader; // initialization

// ... other code

pluginLoader->setFileName("libThatILoad"); // connect with library
```

Etter at biblioteket har blitt knyttet til en QPluginLoader, er den viktigste funksjonen *instance*. [68] Her er dokumentasjonen fra Qts side mangelfull når det kommer til detaljene rundt hva som faktisk skjer. Det som i hvert fall er sikkert er at metoden ved et vellykket kall returnerer en peker til rotkomponenten i biblioteket, og pakker det inn som et QObject (se «Meta-Object-System»). Dette QObjectet må castes til aktuell klassetype, for at bibliotekets innhold (symboler) skal kunne aksesseres. På samme måte som i Nortons metode, vet ikke hovedprogrammet noe om bibliotekets innhold, og kan dermed ikke hente ut klasser direkte. Dermed er en interface-tilnærming også nødvendig ved bruk av QPluginLoader. Vi velger å ikke gjenta denne teknikken her, og henviser til «James Nortons tilnærming til lasting av dynamiske bibliotek i Linux» hvor dette er forklart. For å gjøre om QObjectet til ønsket type, brukes følgende kall:

```
QPluginLoader* pluginLoader("libThatILoad"); // initialize a QPluginLoader with
"libThatILoad"

QObject* temporary_object = pluginLoader->instance(); // returns root component of
the plugin

// cast temporary_object to PluginInterface, and assign to myPlugin:
PluginInterface* myPlugin = qobject_cast<PluginInterface*>(temporary_object);

// use myPlugin like regular objects:
myPlugin->DoSomething();
```

Når man er ferdig med å bruke et bibliotek, kan det lastes ut med *unload*-metoden (se under). På samme måte som for de tidligere utlastningsmetodene, lastes ikke biblioteket ut før alle instanser av QPluginLoader med tilknytning til det biblioteket har benyttet *unload*-metoden. Dette impliserer også at rotkomponenten ikke slettes før siste *unload*-metode er kalt, og at rotkomponenten slettes når siste *unload*-metode kalles.

```
QpluginLoader->unload(); // unload
```

Med disse enkle kallene – kombinert med interface – klarer vi å laste inn klasser på en plattformuavhengig måte. Vi har ikke greid å komme under panseret på QPluginLoader, men vi antar at den er basert på QLibrary. Qt opplyser selv om følgende forskjeller mellom QPluginLoader og QLibrary [67]:

- QPluginLoader fungerer *kun* på Qt Plugins. En Qt Plugin er en plugin som er kompatibel med Qt-rammeverket (se «programvareutviklingsrammeverket Qt). QLibrary har vi derimot demonstrert at også fungerer på øvrige C/C++-biblioteker, så lenge deklarasjonskravene (se forrige delkapittel) av symbolene er dekket.
    - En følge av QPluginLoaders tilknytning til Qt Plugin, er at den kan sjekke om bibliotekets Qt-versjon stemmer overens med hovedprogrammets
- QPluginLoader gir direkte tilgang til et rotkomponentobjekt (QObject) i biblioteket gjennom *instance*-metoden. Dette objektet kan castes til ønsket type, og deretter aksesseres som vanlige objekter i C++. Man slipper altså å måtte løse symboler manuelt ved bruk av *resolve*-metoden.
- QPluginLoader kan, i motsetning til QLibrary, ikke brukes til statisk linking

Vår bruk av QPluginLoader

Ettersom programtilleggene vi skulle laste inn i GeoMOD ville være Qt Plugins – de benytter Qt-funksjonalitet direkte, valgte vi å bruke QPluginLoader. Denne benyttes i *pluginFactory.h* (se «Designmønstre og programvarearkitektur – Abstract factory patterm») I vår implementasjon har vi valgt å bruke én QPluginLoader per bibliotek. Vi har lagret disse i en privat liste (kun tilgjengelig internt i *pluginFactory*), kalt *pluginLoaders_*:

```
QList<QPluginLoader*> pluginLoaders_;
```

Vi valgte å kun bruke én QPluginLoader per bibliotek for å lettere kunne holde oversikt over innlastede bibliotek, og dermed også enklere kunne laste ut bibliotek om nødvendig. Vi ordnet dette med å lage en metode, *getPluginLoader_(QString filePath)*, som returnerer en QPluginLoader til biblioteket på *filePath*s adresse. Hvis ikke det allerede eksisterer en QPluginLoader som er tilknyttet biblioteket, oppretter metoden en ny, og returnerer denne:

```
QPluginLoader* getPluginLoader_(QString filePath)
{
    for(QPluginLoader* loader : pluginLoaders_)
    {
        if(loader->fileName() == filePath) return loader; //
QPluginLoader::fileName()
    }                                                     // gives file path

    // if no QPluginLoader connected to this file exists, create and connect:
    if(QLibrary::isLibrary(filePath))
    {
        addQPluginLoader_(filePath); // create & add new to pluginLoaders__
        return pluginLoaders_.back();
    }

    else return nullptr;
}
```

hvor *addQPluginLoader_* er definert som følger:

```
/// Adds a QPluginLoader to the pluginLoaders-list:
void addQPluginLoader_(QString filePath)
{
    // Check if it already exists:
    for(QPluginLoader* loader : pluginLoaders_)
    {
        if(loader->fileName() == filePath) return;
    }
     // create a new QPluginLoader
    if(QLibrary::isLibrary(filePath))
    {
        QPluginLoader* loader = new QPluginLoader(filePath);
        pluginLoaders_.append(loader);
    }
}
```

Hvert bibliotek som skal lastes inn av QPluginLoader har en klasse som inneholder en metode kalt *newInstance*. Denne metodens oppgave er å returnere et objekt av klassens type. Dette går an, ettersom typen metoden returnerer er definert i interfacet som deles av hovedprogrammet (GeoMOD) og biblioteket. Før vi går nærmere inn på hvordan *newInstance*-metodene virker, skal vi se hvordan de kan nås fra hovedprogrammet ved hjelp av QPluginLoader.

Ved å bruke QPluginLoaders *instance*-metode, kan vi få tilgang til et rotkomponent-objekt av biblioteket. Ettersom biblioteket er en Qt Plugin, vil rotkomponent-objektet i utgangspunktet være av typen QObject. Når rotkomponent-objektet instansieres, kalles konstruktøren i biblioteket. Dette resulterer i at QObjectet vil inneholde alle klassens symboler. Disse kan dog ikke aksesseres enda, fordi hovedprogrammet ikke kjenner til rotkomponent-objektets innhold, da det fortsatt framstår som et QObject. Ved å caste rotkomponent-objektet fra QObject til typen definert av det felles interfacet, kan symbolene benyttes, deriblant *newInstance*.

newInstance-metodene fungerer som følger – her illustrert med kode fra klassen *TestDLL*.

```
PluginInterface* TestDLL::newInstance()
{
    QObject* instance = new TestDLL();

    return qobject_cast<PluginInterface*>(instance);
}
```

TestDLL-klassen arver QObject. Derfor kan man si at TestDLL *er* en QObject. Derfor er kallet:

```
QObject* instance = new TestDLL();
```

tillatt. *TestDLL()* er klassens konstruktør. Når denne kalles, opprettes det et nytt TestDLL-objekt, innlemmet i et QObject. Vi har da to situasjoner hvor klassens konstruktør kalles: Første situasjon vil være når QPluginLoaders *instance*-metode brukes *for første gang* for å opprette et rotkomponent-objekt. Den andre situasjon er hver gang *newInstance* kalles for å opprette nye objekter. En del av bibliotekene gjør mange innstillinger og operasjoner i konstruktøren, som for eksempel å tegne figurer, eller å åpne widgets. Vi ønsker at disse operasjonene *kun* skal bli utført i situasjonene hvor *newInstance* blir kalt, og ikke i forbindelse med opprettelse av rotkomponent-objektet. Dette har vi løst ved å implementere konstruktørene på følgende måte, igjen illustrert med *TestDLL*-klassen:

```
static bool isRootComponent = true;

TestDLL::TestDLL() // CONSTRUCTOR
{
    if(isRootComponent) // first time constructor is called:
    {
        //... no setup
        isRootComponent = false;
    }
    else // called from newInstance:
    {
        //... perform setup
    }
}
```

Til slutt i *newInstance*-metoden, castes *instance* (se kodesnutt på toppen av siden) til typen *PluginInterface* og returneres.

Denne måten å instansiere objekter fra bibliotekene har i praksis vist seg å fungere ypperlig. Det at den gjerne er mer lettfattelig og lesbar enn Nortons metode, ved at en del intrikate operasjoner gjøres under panseret, kombinert med plattformuavhengigheten Qt-verktøyene tilbyr, gjør løsningen fullverdig, sett i forhold til prosjektbeskrivelsens krav.

HVORDAN OPPRETTE QT PLUGINS

Qt Plugins er simpelthen programtillegg som bruker Qt-funksjonalitet. Slike programtillegg kan derfor kun brukes sammen med annen Qt-programvare. En fordel med programtillegg av denne typen, er at de kan lastes inn i programmet av QPluginLoader, som ble beskrevet i forrige delkapittel. Dette verktøyet gjør innlastingsprosessen mer brukervennlig, da den gir tilgang til et rotkomponent-objekt av programtillegget, istedenfor at symboler må løses manuelt. Vi vil i dette delkapittelet gi en forklaring på hvordan man kan opprette Qt Plugins.

Dynamiske Qt Plugins

Qt Plugins lastet inn ved hjelp av QPluginLoader bruker, som tidligere nevnt, en interface-tilnærming. Interfacet blir implementert av en eller flere klasser i bibliotek-prosjektet, i tillegg til å bli lagt til i applikasjons-prosjektet for å løse symboler ved kall. Interfacet må også inneholde en makro-instruksjon som sier til Meta-Object Compiler at prosjektet inneholder et interface. Meta-Object compiler er, som beskrevet i delkapittelet «Meta-Object System», en del av Meta-object System som har som formål å utvide funksjonaliteten til et programmeringsspråk. Makroen som skal inkluderes er *Q_DECLARE*_INTERFACE og legges til utenfor interface-deklarasjonen. Denne makroen definerer hjelpefunksjoner for å muliggjøre dynamisk casting fra QObject til interface-typen [69]. Den første parameteren i makro-instruksjonen er navnet på grensesnitt-filen og den andre strengen er en identifikator. Et eksempel på et slikt interface er avbildet under.

```
class PluginInterface{
    virtual void DoSomething() const = 0;
};

Q_DECLARE_INTERFACE(PluginInterface, "org.examples.PluginInterface.1")
```

Klassen som implementerer interfacet i Qt Plugin-applikasjonen må også implementere QObject. Dette er viktig for at applikasjonen som laster inn biblioteket skal kunne kalle *instance*-metoden til *QPluginLoader*, som returnerer et objekt av typen QObject (se forrige delkapittel). To makro-instruksjoner må også legges til i starten av klassedefinisjonen i header-filen til den implementerende klassen. *Q_OBJECT*-makroen er nødvendig for at Meta-Object System skal lage en C++-fil med Meta-Object-kode av klassen (se «Meta-Object System»). *Q_INTERFACES* forteller Meta-Object System hvilke grensesnitt klassen implementerer.

```
class MyPlugin : public QObject, public PluginInterface
{
    Q_OBJECT
    Q_INTERFACES( PluginInterface )

public:
    //public methods...
};
```

For å spesifisere at prosjektet skal kompileres og linkes som et dynamisk-bibliotek-prosjekt, må også prosjektfilen (prosjektets *.pro*-fil) endres. I filen må *TEMPLATE*-variabelen spesifiseres til *lib*. Dette forteller *qmake* at den skal generere en *Makefile* av prosjektfilen for å bygge biblioteker. qmake er et verktøy i Qt som forenkler bygge-prosessen ved utvikling over forskjellige plattformer. En Makefile er en fil som inneholder instruksjoner for hvordan man kompilerer og linker et program. *CONFIG*-variabelen blir satt til *plugin*. Dette medfører at output etter kompilering blir en *.dll*-fil i Windows og en *.so*-fil i Linux. Den er også nødvendig på noen plattformer for å unngå at symbolske lenker (fil på operativkjerne-nivå som er en referanse til et annet sted i mapperhierarkiet [70]) får versjonsnummer i filnavnet (se kapittel «Programvarebibliotek – Dynamiske bibliotek – Linux»).

```
TEMPLATE = lib
CONFIG += plugin
```

Statiske Qt Plugins

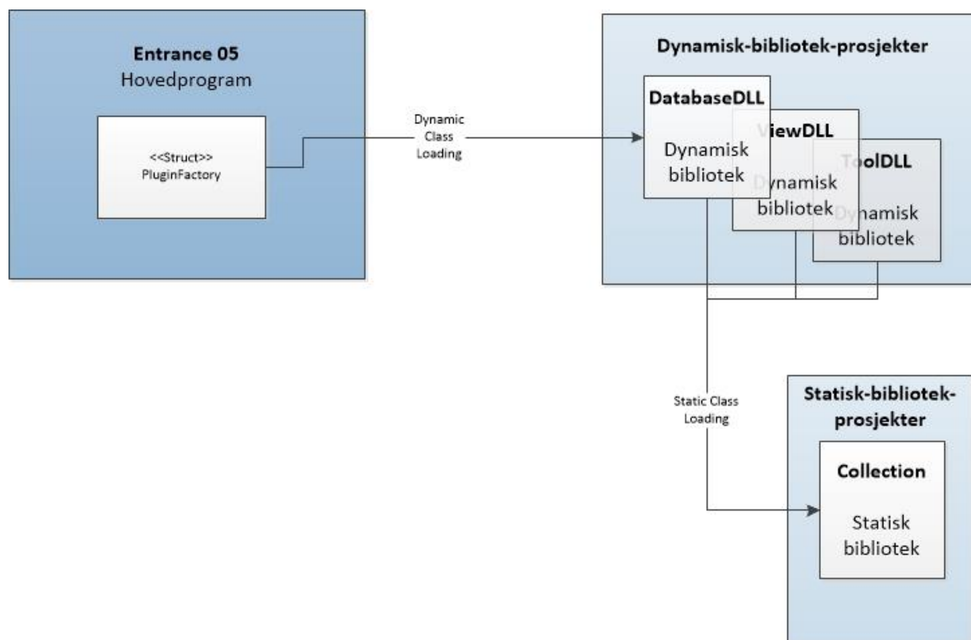
I motsetning til dynamiske bibliotek, trenger man ikke å tilrettelegge for eksport og import av symboler i et statisk bibliotek i Qt. For statiske bibliotek blir, som tidligere nevnt i «Programvarebibliotek – Statiske bibliotek», dette løst under kompilering og linking. I et statisk-bibliotek-prosjekt må man kun spesifisere *TEMPLATE*- og *CONFIG* -variabel i *.pro*-filen til henholdsvis *lib* og *staticlib*. Den sistnevnte variabelen medfører et statisk bibliotek som output. De riktige kompilator-instruksjonene blir da automatisk lagt til i Makefilen ved hjelp av qmake.

```
TEMPLATE = lib
CONFIG += staticlib
```

VÅRE PROGRAMVAREBIBLIOTEK

GeoMOD støtter dynamiske programvarebibliotek av tre typer; *Database*, *View* og *Tools*. *Database*-bibliotekene inneholder informasjon om geometrien til modeller, og består av en samling vinduer for å manipulere og styre disse. Brukeren har mulighet til å endre geometrien til modellen og å flytte den rundt i et koordinatsystem ved hjelp av et styringspanel. Modellen blir ikke vist på skjermen før et dynamisk bibliotek av typen *View* også er lastet inn i hovedprogrammet. *View*-biblioteket bruker informasjon fra *Database*-modellen for å vise den frem i et koordinatsystem. Brukeren har mulighet til å endre hvordan synspunktet på modellen skal være, enten det er ved å følge et sfærisk overblikk, todimensjonalt overblikk eller et fritt synspunkt, hvor brukeren selv bestemmer hvor i koordinatsystemet modellen skal bli sett fra. Den siste typen bibliotek, *Tools*, inneholder funksjonalitet for styring av modeller i baner, samt hvordan *View*-bibliotekene skal virke sammen med *Database*-bibliotekene.

Alle disse dynamiske bibliotekene benytter et statisk bibliotek kalt Collection (se «Extensibility pattern – Plug-in pattern»). Dette biblioteket inneholder matematiske metoder for geometrisk modellering. De dynamiske bibliotekene bruker metoder i Collection for å lage modeller, vise frem modeller og kalkulere og demonstrere styring av modellene i baner. Figur 10 viser en overordnet arkitektur av hvordan bibliotekene henger sammen.



Figur 10 - Overerodnet arkitektur

COLLECTION

Det statiske biblioteket, Collection, består av en samling matematiske og geometriske støtteklasser til bruk i de dynamiske bibliotekene. Filene var ferdigutviklet ved overtagelse av oppgaven, men mye av koden var basert på utdatert og utilgjengelig Qt 3-funksjonalitet. Mye arbeid gikk derfor med til portering til Qt 5.

Filene var samlet i en innfløkt struktur av mapper med filer som var avhengig av hverandre. Mappene kunne deles opp i tre hovedmapper; *Geometry*, *Topology* og *Models*, samt en rekke andre mapper bestående av et varierende antall filer. Geometry inneholdt filer med matematiske klasser for endring av geometrien til modeller og synsvinkler (*Views)*. Topology bestod av filer med funksjonalitet for hvordan hjørner, sidekanter og flater henger sammen. Den siste hovedmappen, Models, inneholdt filer som knytter filene i de to foregående mappene sammen. Vår umiddelbare løsning var å lage statiske bibliotek av hver hovedmappe (Geometry, Topology og Models), og et statisk bibliotek av de øvrige filene samlet i en mappe kalt *shared*. Det viste seg å dukke opp problemer knyttet til rekkefølge av innlasting av bibliotekene. Denne problemstillingen er forklart ytterligere i kapittelet «Videre arbeid».

Løsningen ble å samle filene i *ett* felles statisk bibliotek. Biblioteket blir så lastet inn i hvert dynamiske bibliotek. Fordeler og ulemper ved denne løsningen er diskutert i «Programvarebibliotek - Statiske og dynamiske bibliotekers bruksområder».

DYNAMISKE TESTBIBLIOTEK

Et naturlig steg i prosessen for å få lastet inn dynamiske bibliotek i hovedprogrammet, *Entrance*, var å utvikle enkle dynamiske bibliotek til testformål. Dette ga oss verdifull informasjon om hvorvidt hovedapplikasjonen hadde kontakt med de dynamiske bibliotekene.

Vi utviklet først et testbibliotek med en offentlig metode kalt *DoSomething*, som skrev ut en melding til konsollen (se kodesnutt under). Kall til denne metoden fra hovedapplikasjonen viste at vi hadde kontakt med biblioteket, og at QPluginLoader fungerte som den skulle. Vi brukte også dette biblioteket for å teste om det var mulig å lage flere instanser av typen *TestDLL*. Vi testet dette ved å kalle DoSomething-metoden på hvert av objektene som ble instansiert. Fremgangsmåten for å lage flere instanser av samme bibliotek blir gjennomgått i «Programvarebibliotek i Qt - Vår bruk av QPluginLoader».

```
void TestDLL::DoSomething()
{
    std::cout <<  "Message from DLL" << std::endl;
}
```

OPPDATERING AV PROGRAMVAREBIBLIOTEK UNDER KJØRETID

Ved utvikling og feilsøking av dynamiske programvarebibliotek, vil det være en fordel å kunne laste inn flere versjoner av samme DLL-fil i hovedprogrammet. Dette vil gi utvikleren mulighet til å sammenlikne en tidligere versjon av programvarebiblioteket med nyere versjoner. Vi var usikre på om dette var mulig med Qt, og valgte derfor å lage et testbibliotek for å undersøke dette nærmere.

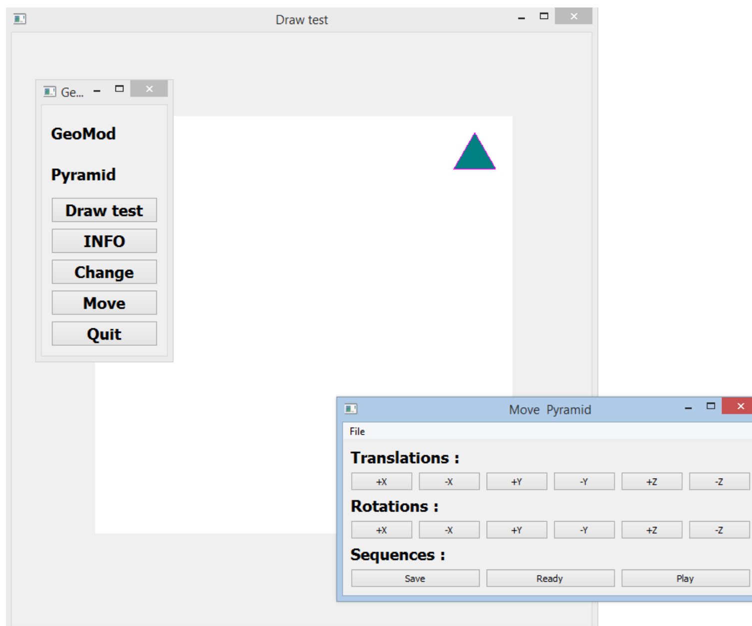
Kompilering av et dynamisk programvarebibliotek vil gi en DLL-fil som output. Gjør man deretter noen endringer i programvarebiblioteket og kompilerer på nytt, vil dette overskrive den opprinnelige DLL-filen. Med *QPluginLoader* viste det seg å ikke være mulig å laste inn en ny versjon av biblioteket med samme filbane, uten å laste ut den tidligere versjonen av biblioteket først. For å sammenligne og interagere med instanser fra en nyere og eldre versjon av samme bibliotek, må derfor objektene fra det gamle biblioteket fortsette å eksistere i minnet, selv etter biblioteket er lastet ut. Vi implementerte en ekko-test, for å undersøke dette videre.

Ekko-test-biblioteket opprettet et grafisk brukergrensesnitt med en knapp som var tilknyttet en funksjon som skrev ut en melding til konsollen. Hensikten med testen var å opprette et objekt av test-biblioteket, laste ut biblioteket, og deretter undersøke om ekko-knappen fremdeles kunne skrive ut en melding til konsoll eller ikke. Det viste seg at det *ikke* ble skrevet ut melding til konsoll etter at biblioteket var lastet ut. Vi konkluderte med at også objektet ble fjernet fra minnet når biblioteket ble lastet ut av hovedprogrammet.

En måte å håndtere dette problemet på, er å legge til versjonsnummer på DLL-filene. Ved å spesifisere versjonsnummer i bibliotekets prosjektfil, vil den resulterende DLL-filen få dette nummeret som suffiks til sitt ordinære biblioteknavn. En praktisk test vi gjennomførte, viser at dette er uproblematisk. Altså kan man laste inn identiske DLL-filer, skulle man ønske det, så lenge de ikke deler filbane/navn. Det vil på samme måte være mulig å laste inn nye versjoner av et bibliotek under kjøretid, så lenge filnavnet/filbanen er forskjellig fra det allerede innlastede bibliotekets.

PYRAMID_01

Pyramid_01 var det første GeoMOD-relaterte dynamiske biblioteket vi arbeidet for å få lastet inn. Dette biblioteket var av typen *Database*, og opprettet en pyramidemodell ved hjelp av diverse klasser og metoder i Collection. I tillegg til å instansiere geometrien til en pyramidemodell, inneholdt biblioteket et vindu for styring av modellen i et koordinatsystem. Vinduet er generelt for alle *Database*-modeller, og kan derfor brukes av andre *Database*-bibliotek. Pyramide_01-biblioteket var et påbegynt prosjekt, men vinduene var ikke ferdigutviklet, og prosjektet var ikke satt opp til å bli kompilert som et dynamisk bibliotek i Qt. Fremgangsmåten vi brukte for å kompilere og strukturere prosjektet som et dynamisk bibliotek, blir beskrevet i detalj i kapitelene «Programvarebibliotek i Qt – Hvordan opprette Qt Plugins».



Figur 11 - Pyramid_01 – Velkomsvindu, «Draw test»-vindu og styringspanel

Database-bibliotekene vil, som sagt, ikke vise noen modeller på skjermen før et dynamisk bibliotek av typen *View* også er lastet inn i hovedprogrammet. Av tidsmessige årsaker utviklet vi kun et *Database*-bibliotek, og fikk dermed ikke kontrollert om tegnemetodene i Collection fungerte ved hjelp av et *View*-bibliotek. For å undersøke dette, lagde vi en test direkte i Pyramide_01-prosjektet. Vi la til en knapp på velkomstvinduet kalt «Draw test», som åpnet et vindu Collection kunne bruke som lerret. Her fikk vi til å tegne en geometrisk figur, samt å fylle den med en farge ved hjelp av tegnemetoder i Collection. Vi hadde dermed vist at tegnefunksjonaliteten fungerte, uten å implementere et bibliotek av typen *View*. Figur 11 viser en skjermdump av velkomstvinduet, «Draw test»-vinduet og styringspanelet i Pyramide_01-prosjektet.

GRAFISK BRUKERGRENSESNITT

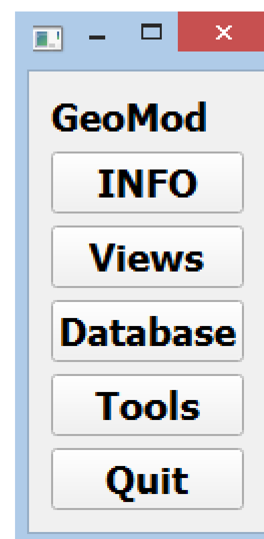
Den visuelle komposisjonen og oppførselen til et grafisk brukergrensesnitt er en viktig del av menneske-maskin-komponenten innen systemutvikling. Målet med et godt brukergrensesnittdesign er å maksimere brukeropplevelsen ved å gjøre brukerens interaksjon så lettfattelig og effektiv som mulig[71]. Brukeren interagerer med bakomliggende data ved å manipulere widgets (knapper, glidebrytere, tekstfelt og lignende). Det er derfor viktig å benytte widgets som gir brukeren mulighet til å oppnå alle mål hun skulle ha. Vårt inntrykk av programmet da vi overtok prosjektet, var at det ikke tilbudte et godt nok brukergrensesnitt. Vi bestemte oss derfor for å re-designe noen av de opprinnelige grafiske brukergrensesnittene. Designmønsteret Model-View-Controller (se «Designmønstre og programvarearkitektur – MVC – Model-view-Controller») tillot oss, ved separasjon av programdelene, å gjøre endringer på grensesnittet uten at det førte til endringer i datahåndteringen.

De grafiske brukergrensesnittene er utviklet i tråd med Qts Object Model, beskrevet i kapittelet «Meta-Object System». Klassene arver fra *QWidget* eller *QMainWindow*. Dette er klasser som tilbyr måter strukturere vinduer. QMainWindow arver fra QWidget og skiller seg fra QWidget ved å tilby enklere måter å implementere rullegardinmenyer. Klassene spesifiserer hvilke objekter som er foreldre-objekter, og kan dermed kommunisere med hverandre direkte. Dette har vi tatt i bruk ved kommunikasjon mellom de ulike manager-vinduene; *View-*, *Database-* og *ToolsManager*, og filutforsker-vinduet. Hver av de tre manager-klassene kan opprette en instans av filutforsker-klassen og motta informasjon om filbaner, ved at filutforsker-klassen sender informasjonen til foreldreklassen, uten å spesifisere på forhånd hvilken av de tre manager-klassene som er forelderen. Dette er mulig ved at manager-klassene implementerer et felles interface (ikke(!) et brukergrensesnitt) med en felles metode for mottak av informasjon fra filutforsker-klassen. Vi kommer tilbake til manager-vinduene og filutforskeren senere i dette kapittelet.

Signals og slots er også en funksjonalitet som har vært nyttig i utviklingen av grafiske brukergrensesnitt. Denne funksjonaliteten tilbyr, som beskrevet i «Programvareutviklingsverktøy - Meta-Object System», en måte å automatisk kalle metoder som følge av brukerinteraksjon med widgets.

OPPRINNELIG DESIGN

Tanken bak det opprinnelige designet av brukergrensesnittet var at hver funksjonelle del av programmet skulle ha sitt eget vindu. Trykker man eksempelvis på «INFO» på velkomstvinduet i *Entrance*, avbildet i figur 12, vil dette resultere i et nytt og separat vindu som viser seg øverst, mens de andre vinduene fortsetter å eksistere i bakgrunnen. Brukeren har dermed selv makt over hvilke vinduer som skal maksimeres, minimeres eller lukkes.

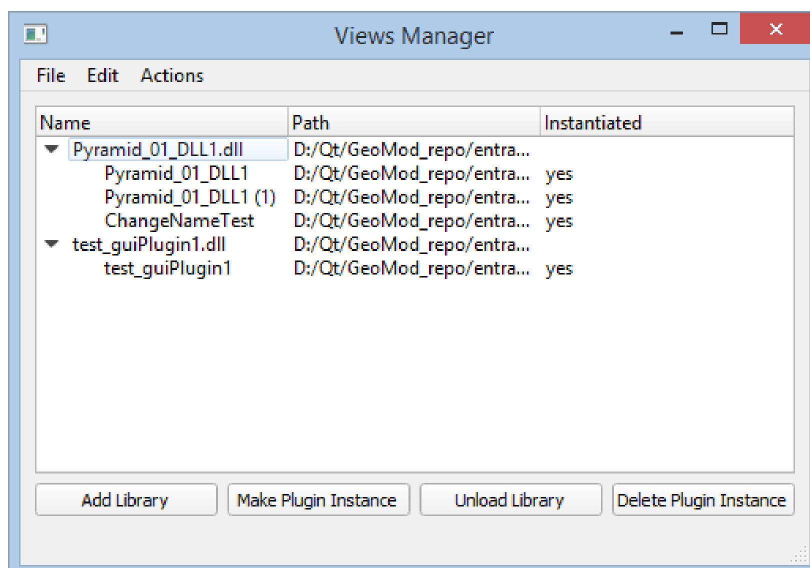


Figur 12 – Entrance Velkomstvindu

Et alternativ til denne typen design er å koble alle brukergrensesnitt sammen i én ramme. Dette gjør det blant annet lettere å holde oversikt over mange forskjellige applikasjoner på en gang, ved at du bare vil ha ett vindu å holde styr på. Baksiden ved denne fremgangsmåten er at brukeren ikke får like mye frihet til å tilpasse komposisjonen av vinduene selv. Vi ser på dette som viktig, da det bagatelliserer problemer ved skalering og tilpasning til forskjellige skjermstørrelser. Vi valgte derfor å beholde den opprinnelige strukturen på brukergrensesnittene, og heller fokusere på hvordan vi kunne forbedre hvert enkelt vindu.

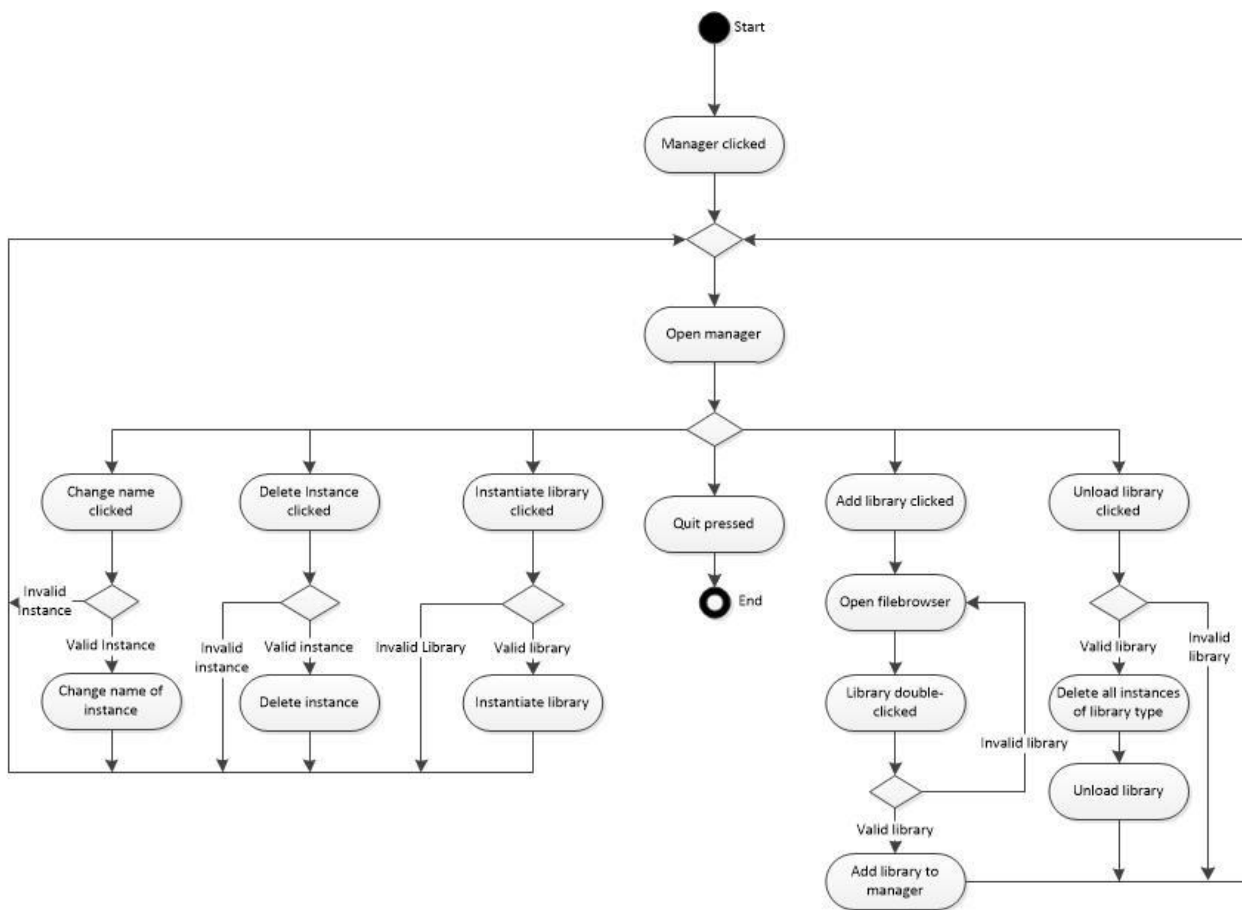
MANAGER-VINDUENE

Manager-vinduene (*ViewsManagerWidget, ToolsManagerWidget, DataBsManagerWidget*) hadde størst rom for forbedring. Ved overtagelse av prosjektet var det utviklet en halvferdig og ikke-fungerende løsning basert på Qt sine egne metoder for å bygge en trestruktur. Vi ferdigstilte den påbegynte løsningen ved å bruke en trestruktur med



Figur 13 - ViewsManagerWidget

programtilleggene som foreldre-node, og instanser av programtillegg som barn-node. Denne visningen gjør det enkelt å skille mellom typer av instanser, samt å holde oversikt over hvor mange instanser som har blitt opprettet. For å øke brukervennligheten implementerte vi også lett tilgjengelige widgets for å endre navn på instansene, oppretting og sletting av instanser, og inn- og utlasting av bibliotek. Figur 13 viser en skjermdump av ViewsManagerWidget. Tilstandsdiagrammet på figur 14 illustrerer valgene brukeren har, og tilstanden valgene fører til i manager-vinduene.

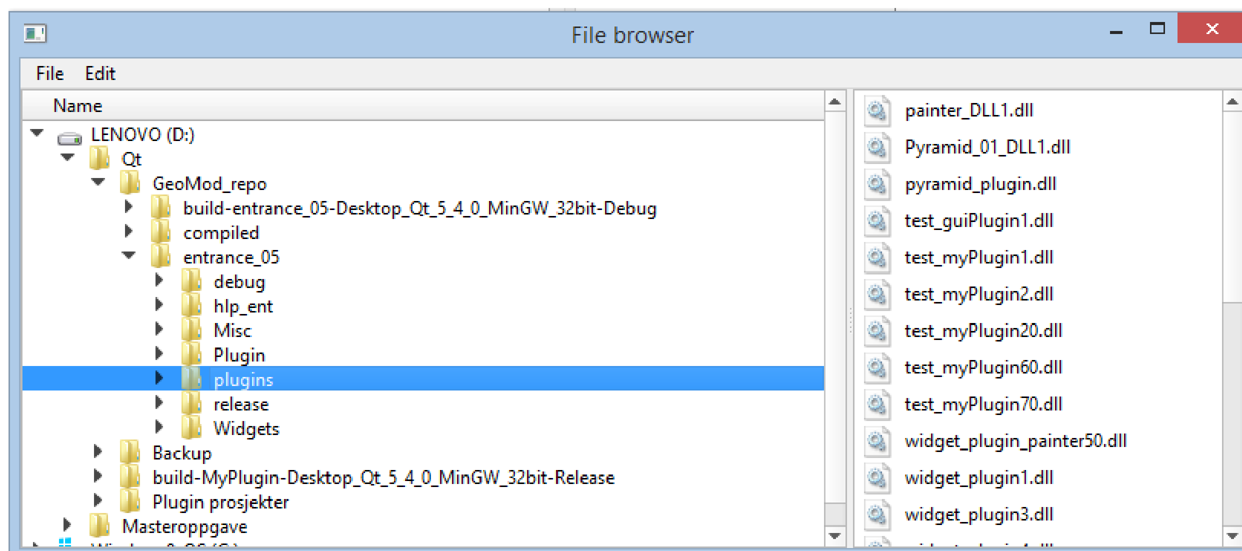


Figur 14 – Tilstandsdiagram

FILUTFORSKER-VINDUET

Filutforsker-vinduet var satt opp med tradisjonell Windows-stil; en trestruktur med mapper på venstre side, og en listestruktur med filer i de valgte mappene på høyre side. Visningen av mapper og filer hadde et forbedringspotensial ved at vinduet var for lite til å vise hele filbaner. Vi utvidet derfor størrelsen på vinduet, samt justerte forholdet mellom størrelsen på mappe- og filvisning, slik at

brukeren ikke trenger å justere dette selv. Siden applikasjonen kun kan laste inn dynamiske bibliotek i kjøretid, var det også overflødig å vise alle tilgjengelige filer i listestrukturen. For å gjøre visningen av filer så enkel og effektiv som mulig, la vi til et filter som gjorde at kun programtillegg vises (dll/so-filer). Dette filteret kan også merkes av i *edit*-menyen. Figur 15 viser en skjermdump av filutforsker-vinduet.

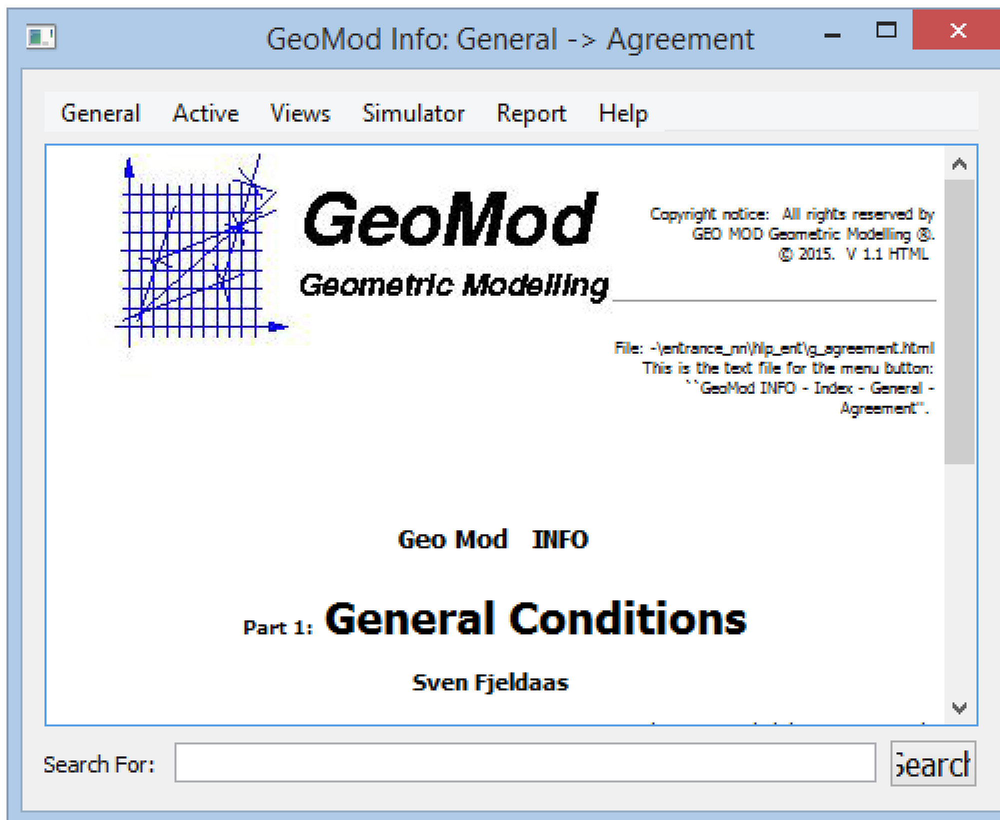


Figur 15 - File browser

INFO-VINDUET

Info-vinduet var ved overtagelse av prosjektet ikke fungerende. Info-vinduet skulle lese og vise tekstfiler med hjelpeinformasjon som lå lokalt på maskinen. Det var også et ønske om å vise figurer i info-vinduet. Filene var skrevet på *html*-format, men var lagret på *txt*-format. Html er et markeringsspråk for formatering av nettsider med informasjon som vises i en nettleser [72]. Med markeringsspråk menes et system for å annotere et tekstdokument slik at det er mulig å skille annoteringene med vanlig tekst [73]. txt-format er et format som ikke tar hensyn til annoteringer, og viser alt som det står skrevet i dokumentet. I implementasjonen av info-vinduet ble hver enkelt linje i txt-filen lest inn og senere forsøkt tolket av en Qt-html-leser kalt *QTextEdit*. Løsningen tok ikke vare på all formatering, og viste heller ikke figurene lagt til i markeringsspråket. Vi valgte derfor å implementere innlesingsmekanismen på ny. Vår løsning var å lagre txt-filene på html-format, og deretter lese inn filene ved hjelp av en annen Qt-html-leser, kalt *QTextBrowser*. QTextBrowser bygger på QTextEdit, men gir i tillegg brukeren mulighet til å trykke på lenker [74]. Html-filen ble så lest

inn ved hjelp av innebygde metoder i QTextBrowser, og viste frem tekst og figurer, som ønsket. Dette gjorde det ekstra steget med å lese inn hver linje i txt-filen unødvendig. Figur 16 viser hvordan info-vinduet ser ut.



Figur 16 – InfoWidget

Qt tilbyr en enkel måte å endre det tematiske utseendet i applikasjonen ved hjelp av *QApplication* sin *setStyle()*-metode. Det finnes en rekke forskjellige stiler å velge mellom, men i våre øyne var stilen *fusion* den smukkeste. Dette kan gjøres på følgende måte:

```
QApplication::setStyle(QStyleFactory::create("Fusion"));
```

Kompilering, linking og distribuering i Windows

Innenfor Qt Creator

For å kompilere og linke Qt-applikasjoner, må man ha en gyldig versjon av Qt Creator (community) [75] installert på datamaskinen. Vi har benyttet Qt versjon 5,4 med MinGW-kompilator (32-bit). Andre kompilatorer kan også benyttes, men det er kritisk at programtilleggene som skal lastes inn er kompilert med samme kompilatortype som hovedprogrammet. Når Qt er installert på maskinen, kan prosjektet åpnes ved å dobbeltklikke *.pro*-filen i prosjektmappen. Heretter kan man bygge (kompilere og linke) prosjektet ved å trykke på hammerikonet nederst til venstre i Qt Creator. Når kompilatoren har bygget prosjektet ferdig, kan man trykke på play-knappen i samme område som hammerikonet for å kjøre prosjektet.

Windows kommandolinje

For å kompilere, linke og kjøre programmet fra kommandolinjen i Windows, må man ha en kompilator installert på datamaskinen. Som sagt, bruker vi MinGW-kompilatoren. I kommandolinjen navigerer man først til mappen som prosjektet ligger i. Her kjører man *qmake* (beskrevet under) for å lage en *Makefile* fra *.pro*-filen. Deretter kjører man – i vårt tilfelle, med vår kompilator – *mingw32-make* for å kompilere og linke prosjektet. Til slutt kjører man programmet ved å kalle den kjørbare filen som har blitt generert (.exe). Prosessen oppsummeres her (første trinn med å navigere til prosjektmappen er utelatt):

```
qmake makefile MyProject.pro
mingw32-make
C:/path_to_exe/MyProject.exe
```

Distribusjon av Qt-baserte program i Windows

For å distribuere Qt-applikasjonen, må man være sikker på at alle filer som blir brukt av applikasjonen blir lagt ved. Ved siden av den kjørbare filen (.exe), kreves en del DLL- og konfigurasjonsfiler, som må legges i *release*-mappen til applikasjonen. En Qt-applikasjon i Windows refererer dynamisk til mange forskjellige DLL-filer, som blir ordnet opp i automatisk ved installasjon av Qt. Når man skal distribuere en Qt-applikasjon til Windows-PCer hvor Qt ikke er installert, blir ting mer komplisert. Alle bibliotek som blir brukt i applikasjonen må bli distribuert, og det er

spesifikke regler for hvordan de må organiseres i distribusjonskatalogen. DLL-filene som trengs for å kjøre en utgave av applikasjonen, kan deles i tre kategorier: Qt-spesifikke DLLer, C++-kompilatorspesifikke DLLer og DLLer lastet av Qt selv (plugin-moduler) [76]. Under følger en forklaring av disse.

QT-SPESIFIKKE DLLER

Av de Qt-spesifikke DLLene trengs *Qt5Core.dll*, *Qt5Gui.dll* og *Qt5Widgets.dll* i Windows. Ved en vanlig installasjon av Qt ligger disse filene i kompilatoren sin *bin*-mappe. For at operativsystemet skal finne DLL-filene, legges en kopi av disse i samme mappe som den kjørbare applikasjonsfilen.

C++-KOMPILATORSPESIFIKKE DLLER

Avhengig av hvordan Qt er konfigurert, må kompilatorspesifikke DLLer legges ved den kompilerte applikasjonen. I vårt tilfelle bruker vi MinGW-kompilator, og må derfor legge ved *libwinpthread-1.dll*, *libgcc_s_dw2-1.dll* og *libstdc++-6.dll*. For å finne ut hvilke kompilatorspesifikke DLL-filer som mangler, kan man benytte verktøyet *Dependency Walker* [77], som viser hvilke bibliotek en applikasjon eller et annet bibliotek prøver å linke til. Hvis Qt ble konfigurert til å bruke *ICU* [78], må også *icudtXX.dll*, *icuinXX.dll* og *icuucXX.dll* legges ved. ICU (International Components for Unicode) er et bibliotek brukt av Qt for støtte av Unicode-tegnkoding. Disse DLL-filene finnes i /*bin*-mappen av Qt-installasjonen.

DLLER LASTET AV QT SELV

DLLer lastet av Qt selv, også kalt plugin-moduler, skiller seg fra de andre kategoriene ved at de ikke blir lastet inn av operativsystemet, men av Qt selv. Dependency Walker vil her ikke være til hjelp, da dette verktøyet bare bryr seg om DLL-filer etterspurt av applikasjonen, og ikke DLL-filene Qt bruker selv. For å laste inn disse plugin-modulene, begynner Qt å lete i en mappe kalt *plugins*. I denne mappen finnes det 16 undermapper med forhåndsbestemte navn, hvor plugin-modulene må ligge. Den viktigste undermappen er *platforms*. Her ligger *qwindows.dll,* som er nødvendig for at selv den enkleste Qt-applikasjon skal kunne kjøre på Windows. Alle Qt GUI-applikasjoner (Graphical User Interface) trenger en plugin-modul som implementerer *Qt Platform Abstraction (QPA) layer* i Qt 5. QPA er et plattform-abstraksjonslag i Qt 5. Det er ansvarlig for mange operativsystemspesifikke oppgaver, for eksempel å oversette et kall som setter vinduet i applikasjonen til maksimal størrelse. Det er altså viktig at *qwindows.dll* ligger i *platforms*-mappen som legges sammen med de andre filene i *release*-mappen.

Distribusjonen av programmet trenger også de selvlagde dynamiske bibliotekene som blir lastet inn i kjøretid. I vårt tilfelle henter brukeren ut filbanen til disse bibliotekene ved hjelp av en filutforsker, og de kan derfor ligge hvor som helst på PCen. Under følger skjermdumper av hvordan strukturen i release-mappen ser ut.

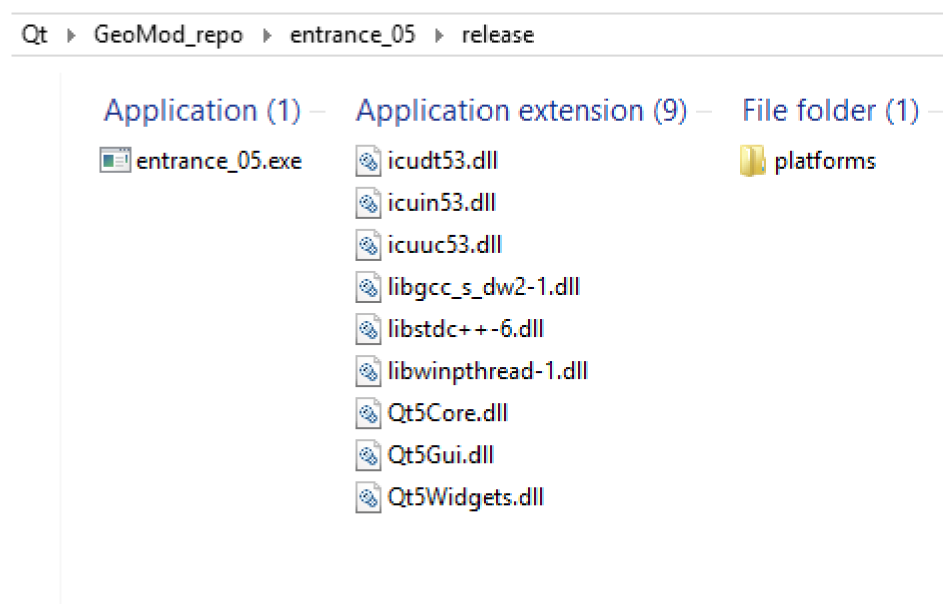


Figure 17 - Distribusjonsmappe

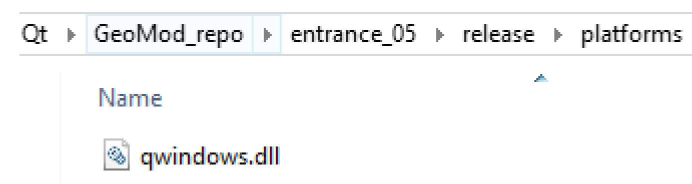


Figure 18 - Innhold i platforms-mappe

OPPSUMMERING OG ANBEFALINGER TIL VIDERE ARBEID

Arbeidet med GeoMOD har pågått i over ti år. Prosjektet har vært arbeidet på i forbindelse med flere prosjekt- og masteroppgaver. Derfor har utviklingen vært noe etappepreget, og ikke så kontinuerlig som en del, mer avgrenset programvare kan være. Det at utviklingen har gått over lang tid, har tvunget oss til å måtte ta hensyn til ytre faktorer som nye programvaretekniske innretninger, deriblant, og mest sentralt, nye versjoner av programvareutviklingsrammeverket Qt. I arbeidet vårt med å oppdatere den gamle versjonen av GeoMOD som vi overtok ved prosjektstart, til å være kompatibelt med den nyeste Qt-versjonen (Qt 5,4), oppstod det en del utfordringer. Store deler av den utdaterte funksjonaliteten kunne erstattes med nye funksjoner direkte, men en del funksjonalitet, deriblant menysystem og filleseren, måtte implementeres på ny.

Som fremkommer innledningsvis i denne besvarelsen, er innlasting av dynamiske programvarebibliotek essensielt for autonome, arbeidende fartøy. Et autonomt fartøy må kunne respondere på endringer i omgivelsene på en adekvat måte. Derfor er det helt essensielt at programmet kan laste inn og ut programtillegg basert på behov som justeres på en kontinuerlig basis, slik at det har funksjonalitet tilgjengelig til å takle uventede utfordringer. På sikt kan det også bli aktuelt at fartøy som benytter GeoMOD kan *sanntidsprogrammeres* [81], det vil si at kildekoden blir skrevet og kompilert av fartøyet selv, basert på påvirkning fra ytre faktorer.

Programmeringsteknisk er operasjoner knyttet til inn- og utlasting av programvarebibliotek ofte intrikate. Det at Qt tilbyr konkrete verktøy med fokus på brukervennlighet, viser nettopp det. Med inn- og utlastingsmekanismen vi har utviklet i prosjektet, kan et fartøy som opererer under vann, og som har behov for å gjøre endringer i et innlastet bibliotek, laste dette biblioteket ut, skrive om kildekoden, rekompilere og deretter laste det inn igjen.

Andre problemstillinger, som vi har vurdert å falle noe utenfor vårt ansvarsområde, har vi funnet provisoriske løsninger for, og overlatt til senere prosjektarbeidere å jobbe videre med. Vi vil i de neste delkapitlene komme med noen anbefalinger til videre arbeid.

KOMMUNIKASJON MELLOM DYNAMISKE BIBLIOTEK

Av tidsmessige årsaker utviklet vi kun ett dynamisk bibliotek, av type *Database*. Dette kommuniserer med det statiske biblioteket Collection, som inneholder fellesmetoder for de dynamiske bibliotekene (se «Våre bibliotek») Det gjenstår å lage et dynamisk bibliotek av typene *View* og *Tools*. Vi har

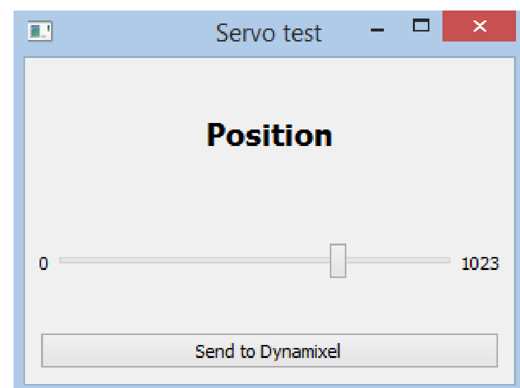
demonstrert at tegnemetodene i Collection fungerer, og utviklingen av *View*-bibliotek bør derfor ikke by på store utfordringer.

Et krav som følger når dynamiske bibliotek skal kommunisere, er å implementere et system av metoder for varsling og oppdatering når innvirkende endringer gjøres. Et eksempel på dette er metode(r) som holder orden på når noe skal tegnes og hva som skal tegnes. Denne kommunikasjonen mellom bibliotekene var ved overtagelse av prosjektet på ad-hoc nivå og er på nåværende tidspunkt utelatt fra vår løsning. Den fungerte ved at hovedprogrammet *Entrance* holdt en oversikt over operative tegneflater og modeller. Flagg i *Entrance* fungerte som signal til bibliotekene om at det er på tide med en oppdatering. De ulike bibliotekene kommuniserte da via containere i *Entrance* som inneholdt pekere til funksjoner i bibliotekene. Vi ser det fornuftig at overtagende prosjektarbeidere forsøker å finne en permanent og god løsning som erstatter ad-hoc-systemet.

KOBLING TIL DYNAMIXEL-AKTUATORER

Det ville vært av verdi å knytte GeoMOD opp mot elektromotor for å demonstrere at systemet kan kalkulere vinkler og sende disse som instruksjoner til en motor. Dette hadde også vært en ypperlig anledning til å knytte oppgaven mot prosjektoppgaven vår, hvor vi utviklet et rammeverk for styring av Dynamixel-servomotorer og -sensorer, med tilhørende grafisk-brukergrensesnitt for styring lokalt og over Internett. Dette rammeverket lagde vi i tre utgaver, en for hvert av programmeringsspråkene Java, C++ og C#. Vi har beskjeftiget oss med å koble C++-versjonen til GeoMOD, men det dukket opp kompilatorspesifikke problemer rundt kombinering av det lavnivå styringssystemet til Dynamixel, som var kompilert med MSVC-kompilatoren (Microsoft Visual C++) og GeoMOD, som kompileres med MinGW-kompilatoren.

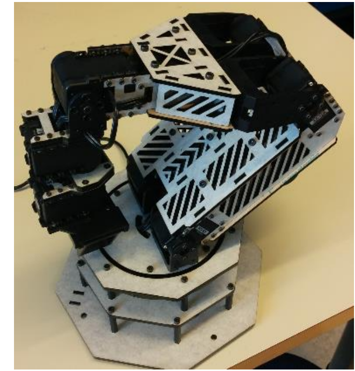
Et naturlig steg i prosessen for å knytte GeoMOD opp mot Dynamixel-aktuatorene var å integrere en styringsflate i GeoMOD for direkte styring av aktuatorene. Vi utviklet denne styringsflaten, avbildet til høyre, og den er en del av *Entrance*-applikasjonen under «Servo»-knappen på velkomstvinduet. Den er ikke tilknyttet implementerte metoder for å sende informasjon videre til Dynamixel-aktuatorene, men er et fint utgangspunkt for videre arbeid. Når aktuatorene er knyttet til GeoMOD – når de



Figur 19 - "Servo test" - vindu

kompilatorspesifikke problemene er løst – bør det være en smal sak å basere kallene til aktuatoren på GeoMODs biblioteker.

Vi har demonstrert styring av Dynamixel-aktuatorene ved å bruke Java-versjonen av styringsprogramvaren vi lagde i prosjektoppgaven, til å utvikle en styringssekvens for PhantomX Reactor Robot Arm [79] som kaster en dart-pil på en dartskive.



Figur 20 - PhantomX Reactor Robot Arm

COLLECTION

Opprinnelig var filene i Collection-biblioteket fordelt på en innfløkt mappestruktur, som også av veileder ble beskrevet som uoversiktlig og uferdig. Dette resulterte i at vi så behovet for en midlertidig forenkling, nemlig å samle alle filene som blir benyttet av majoriteten av de dynamiske bibliotekene i et eget bibliotek, nemlig Collection. Tanken var i første omgang å dele Collection i underbiblioteker, sortert etter ansvarsområder. I forbindelse med linking fra de dynamiske bibliotekene til disse statiske bibliotekene, viste rekkefølgen seg å være av kritisk betydning. Hvis et bibliotek **A** avhenger av symboler definert i et annet statisk bibliotek **B**, må bibliotek **A** lastes inn før bibliotek **B** [80]. Dette førte til et problem ved at filene i de forskjellige mappene var avhengige av hverandre. Løsningen ble å legge alle filene inn i ett felles, statisk bibliotek.

Denne løsningen er ikke ideell, ved at den strider med Single responsibility-prinsippet i SOLID (se «Designmønstre og programvarearkitektur – SOLID – designprinsipper»). Dette prinsippet handler, som tidligere nevnt, om ansvarsavgrensning. Vi så det som fornuftig å gjøre denne midlertidige forenklingen, slik at vi heller kunne fokusere på å få programmet opp og gå. Vi vil påstå at Collection-løsningen tross alt er ryddigere enn den opprinnelige strukturen som var da vi overtok prosjektet.

Å samle de planlagte underbibliotekene i et felles bibliotek har likevel ingen betydning på ytelse. Som beskrevet i delkapittelet «statiske bibliotek - programvarebibliotek», vil kun de aktuelle programdelene hentes ut ved linking. Dette gjør at vi ikke trenger å bekymre oss for å laste unødvendig kode inn i applikasjonene.

KILDER

Innholdet i alle kildene oppført ble verifisert 10.06.2015.

1. History of C++, http://www.cplusplus.com/info/history/
2. Pointer (Computer programming), http://en.wikipedia.org/wiki/Pointer_%28computer_programming%29#/media/File:Pointers.svg
3. Opaque pointer, http://en.wikipedia.org/wiki/Opaque_pointer
4. Arv i programmering, http://nn.wikipedia.org/wiki/Arv_i_programmering
5. Class (computer programming), http://en.wikipedia.org/wiki/Class_(computer_programming)#Abstract_and_Concrete
6. C++, http://en.wikipedia.org/wiki/C%2B%2B
7. Facebook (antall brukere), http://en.wikipedia.org/wiki/Facebook
8. Bjarne Stroustrup: How C++ Combats Global Warming, YouTube-intervju gjort av Big Think, https://www.youtube.com/watch?v=FBWeO2HYEc0
9. Qt Company, http://en.wikipedia.org/wiki/Qt_Company
10. Porting to Qt 4, http://doc.qt.io/qt-4.8/porting4.html
11. Std::string, http://www.cplusplus.com/reference/string/string/
12. The Meta-Object System, http://doc.qt.io/qt-4.8/metaobjects.html
13. QObject – Detailed description, http://doc.qt.io/qt-4.8/qobject.html
14. Signals and slots, http://doc.qt.io/qt-4.8/signalsandslots.html
15. Widget(GUI), http://en.wikipedia.org/wiki/Widget_%28GUI%29
16. QObject, http://doc.qt.io/qt-4.8/qobject.html#Q_OBJECT
17. Preprocessor, http://en.wikipedia.org/wiki/Preprocessor
18. Using the Meta-Object Compiler (moc), http://doc.qt.io/qt-5/moc.html#moc
19. Meta Object Compiler (moc), http://tinf2.vub.ac.be/~dvermeir/manual/KDE20Development-html/ch03lev1sec4.html
20. Qt Creator, http://en.wikipedia.org/wiki/Qt_Creator
21. Home of the MinGW and MSYS Projects, http://www.mingw.org/
22. GCC, the GNU Compiler Collection, https://gcc.gnu.org/
23. Design patterns, https://msdn.microsoft.com/en-us/library/ff647483.aspx
24. Single responsibility principle, http://en.wikipedia.org/wiki/Single_responsibility_principle
25. Open/closed principle, http://en.wikipedia.org/wiki/Open/closed_principle
26. Liskov substitution principle, http://en.wikipedia.org/wiki/Liskov_substitution_principle
27. Interface segregation principle, http://en.wikipedia.org/wiki/Interface_segregation_principle
28. Dependency inversion principle, http://en.wikipedia.org/wiki/Dependency_inversion_principle
29. Model-View-Controller- http://no.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#/media/File:ModelViewControllerDiagram2.svg

30. Model/View Programming, http://doc.qt.io/qt-4.8/model-view-programming.html
31. Extensibility pattern, http://en.wikipedia.org/wiki/Extensibility_pattern
32. Design Patterns, https://msdn.microsoft.com/en-us/library/ff647483.aspx
33. Design Patterns Uncovered: The Abstract Factory Pattern, *James Sugrue*, http://java.dzone.com/articles/design-patterns-abstract-factory
34. Templates and Template Classes in C++, *Alex Allain*, http://www.cprogramming.com/tutorial/templates.html
35. An Idiot's Guide to C++ Templates, *Ajay Vijayvargiya*, http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part
36. C++ Templates, http://www.tutorialspoint.com/cplusplus/cpp_templates.htm
37. Use 'class' or 'typename' for template parameters?, http://stackoverflow.com/questions/213121/use-class-or-typename-for-template-parameters
38. MVC vs. Qt Model/View http://stackoverflow.com/questions/5543198/why-qt-is-misusing-model-view-terminology
39. Library (computing), http://en.wikipedia.org/wiki/Library_%28computing%29
40. Virtual Memory, http://www.brokenthorn.com/Resources/images/virtual-memory%5B1%5D.png
41. Memory resident, http://www.webopedia.com/TERM/M/memory_resident.html
42. Virtual memory, http://en.wikipedia.org/wiki/Virtual_memory
43. Static library, http://en.wikipedia.org/wiki/Static_library
44. Static build, http://en.wikipedia.org/wiki/Static_build
45. Linker (computing), http://en.wikipedia.org/wiki/Linker_(computing)
46. Linker.svg, http://en.wikipedia.org/wiki/File:Linker.svg
47. Loader (computing), http://en.wikipedia.org/wiki/Loader_(computing)
48. Dynamic linker, http://en.wikipedia.org/wiki/Dynamic_linker
49. Dynamic-link library, http://en.wikipedia.org/wiki/Dynamic-link_library
50. Dynamic-Link Library Creation, https://msdn.microsoft.com/en-us/library/windows/desktop/ms682592(v=vs.85).aspx
51. Dynamic loading, http://en.wikipedia.org/wiki/Dynamic_loading
52. How does the Import Library work? Details?, http://stackoverflow.com/questions/3573475/how-does-the-import-library-work-details
53. Intro to Linux Shared Libraries (How to Create Shared Libraries), http://www.thegeekstuff.com/2012/06/linux-shared-libraries/
54. Shared Librarires, http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html
55. Position-independent code, http://en.wikipedia.org/wiki/Position-independent_code
56. C++ instantiate template class from DLL, http://stackoverflow.com/questions/14138360/c-instantiate-template-class-from-dll
57. General Rules and Limitations, https://msdn.microsoft.com/en-us/library/twa2aw10.aspx

58. Dynamic Class Loading for C++ on Linux, http://www.linuxjournal.com/article/3687?page=0,0
59. «The Open Group», <dlfcn.h>, http://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html
60. dlopen(3) – Linux man page, http://linux.die.net/man/3/dlopen
61. OOP PolymorphismAbstractShape, https://www3.ntu.edu.sg/home/ehchua/programming/java/images/OOP_PolymorphismAbstractShape.png
62. Map, http://www.cplusplus.com/reference/map/map/
63. QLibrary Class, http://doc.qt.io/qt-5/qlibrary.html
64. QLibrary, resolve, http://doc.qt.io/qt-5/qlibrary.html#resolve
65. QFunctionPointer QLibrary::resolve(const char* symbol), http://doc.qt.io/qt-5/qlibrary.html#resolve
66. QLibrary::unload(), http://doc.qt.io/qt-5/qlibrary.html#unload
67. QPluginLoader, Detailed Description, http://doc.qt.io/qt-5/qpluginloader.html#details
68. QPluginLoader #instance, http://doc.qt.io/qt-5/qpluginloader.html#instance
69. Plugins, https://wiki.qt.io/Plugins
70. Symbolsk lenke, http://no.wikipedia.org/wiki/Symbolsk_lenke
71. User Interface Design, http://en.wikipedia.org/wiki/User_interface_design
72. HTML, http://no.wikipedia.org/wiki/HTML
73. Markeringsspråk, http://no.wikipedia.org/wiki/Markeringsspr%C3%A5k
74. QTextEdit Class, http://doc.qt.io/qt-4.8/qtextedit.html
75. Dowload Qt, http://www.qt.io/download/
76. Deploying a real Qt app – understanding more of Qt, *hskoglund*, http://www.tripleboot.org/?p=536
77. Dependency Walker, http://www.dependencywalker.com/
78. ICU Home Page, http://site.icu-project.org/
79. PhantomX Reactor Robot Arm Kit, http://www.trossenrobotics.com/p/phantomx-ax-12-reactor-robot-arm.aspx
80. Why does the order in which libraries are linked sometimes cause errors in GCC?, *Johannes Schaub,* http://stackoverflow.com/questions/45135/linker-order-gcc
81. Real-time computing, http://en.wikipedia.org/wiki/Real-time_computing