Wiki »

Exercise 9 - Message System ¶

Motivation

I denne opgave vil det tidligere implementeret OS API blive videreudviklet så det gør brug af et MDS (MessageDistributionSystem), og herved gøre brug af tre forskellige Patterns, Singleton, Mediator og Publisher/Subscriber. Fordelen ved at bruge sådan et system i sit program er, at man opnår en lav kobling som kan udnyttes til en mulig udgivelse. Det skyldes at senderen ikke behøver at "forholde" sig til, hvem der skal modtage beskeden. Samtidig behøver modtageren heller ikke "bekymre" sig om, hvem den modtager beskeder fra.

Exercise 1 - The Message Distribution System - Intro

I denne opgave vil der blive stiftet bekendskab med et MDS, som skal håndterer beskeder mellem sender og modtager. Der er blevet udleveret et næsten komplet MDS, heri mangler bare implementering i .cpp-filen. Den vil blive implementeret ud fra opgavebeskrivelsen UML-diagram med Subscriber, MDS og Publisher.

Exercise 2 - The Message Distribution System

Exercise 2.1 - Design and implementation

I den udleverede MDS.zip eksisterer der filen *MessageDistributionSystem.hpp*, som der vil blive arbejde med i tre større underopgaver:

Exercise 2.1.1 - Why a template method?

Årsagen til at notify funktionen skal være en template funktion, ligger i at at den besked der medgives som parameter inde i funktionen kopieres. Hvis man blot sendte den samme besked afsted til alle modtagere, var det ikke nødvendigt med en template funktion, da de forskellige besked typer arver fra Message, og derfor godt kan medgives som en pointer parameter i send funktionen. Det ønskes dog at sende en kopi af beskeden afsted i stedet, for hvis flere tråde skal have samme besked, opstår der problematiker ift. ændring af beskeddat, og sletning af besked.
Derfor skal beskeden kopieres. dette gøres i linjen
M *tmp = new M(*m);
Denne linje er årsagen bag template funktionen.
Det er jo essensielt at der oprettes en kopi af den rigtige besked type. Hvis funktionen ikke var en template funktion, ville det være umuligt at vide hvilken copy constructor der skulle kaldes inde i funktionen.

Exercise 2.1.2 - API Implementation

I denne opgave vil de manglende implementationer blive skrevet:

SubscriberId()

```
SubscriberId::SubscriberId(osapi::MsgQueue* mq_, unsigned long id_)
:mq_(mq_), id_(id_)
{
  /* Make your own implementation here... */
}
```

SubscriberId::send()

```
/** Send the message to the subscriber*/
void SubscriberId::send(osapi::Message* m) const
{
  /* What do you do when you want to send to a reciever?
     What do you need to know? */
     mq_->send(id_,m);
}
```

MessageDistributionSystem::subscribe()

```
void MessageDistributionSystem::subscribe(const std::string& msgId,
                                          osapi::MsgQueue* mq,
                                          unsigned long id)
{
  /* Something missing */
  osapi::ScopedLock lock(m_);

  InsertResult ir = sm_.insert(std::make_pair(msgId, SubscriberIdContainer()));
  SubscriberIdContainer& sl = ir.first->second;

  details::SubscriberId s(mq, id);

  SubscriberIdContainer::iterator iter = find(sl.begin(), sl.end(), s);
  if(iter == sl.end())
    sl.push_back(s);

}
```

MessageDistributionSystem::unSubscribe()

```
void MessageDistributionSystem::unSubscribe(const std::string& msgId,
                                            osapi::MsgQueue* mq,
                                            unsigned long id)
{
  // Find SubscriberIdContainer via via iterator
  osapi::ScopedLock lock(m_);
  SubscriberIdMap::iterator iter  = sm_.find(msgId);
  if(iter!=sm_.end())
  {

    details::SubscriberId container(mq,id);
    SubscriberIdContainer& subList = iter->second;
```

```
    for(auto it=subList.begin();it!=subList.end();++it)
    {
      if(*it==container)
      {
        subList.erase(it--);
      }
    }
  }
  // Where else do I need to find this container?
  // The struct std::pair (the element in std::map<>) contains two vars one
  // named first and one named second. (See funktionen std::make_pair() - JFGI)
  // If found. How do you know that an element in a container was not found?

  // Create a details::SubscriberId and use that to find the desired subscriber.
  // When found it must be erased
}
```

Exercise 2.1.3 - Test harness implementation

Til at teste bruges vedlagte test setup, der mangler dog stadigvæk kode i *Subscriber.cpp* og *Publisher.cpp* og er blevet implementeret på følgende måde:

Subscriber.cpp

```
Subscriber::Subscriber(unsigned int subId)
  :  mq_(MAX_QUEUE_SIZE), subId_(subId)
{
  /*******************************************************/
  /*******************************************************/
  /* Write the necessary code to subscribe to an event */
  /*******************************************************/
  /*******************************************************/
  MessageDistributionSystem &instance=MessageDistributionSystem::getInstance();
  instance.subscribe(HELLO_MSG,&mq_,ID_HELLO);

}

void Subscriber::handleMsgIdHello(HelloMsg* hm)
{
  OSAPI_LOG_DBG("S(" << subId_ << ") The hello message contained: '" << hm->data_ <<
  /*******************************************************/
  /*******************************************************/
  /* Write the necessary code to unsubscribe to an event */
  /*******************************************************/
  /*******************************************************/
  MessageDistributionSystem &instance=MessageDistributionSystem::getInstance();
  instance.unSubscribe(HELLO_MSG,&mq_,ID_HELLO);
  running_=false;
}
```

Publisher.cpp

```
void Publisher::handleMsgIdTimeOut()
{
```

```
  OSAPI_LOG_DBG("Got timeout, publishing message and rearming...");
  /*********************************************/
  /*********************************************/
  /* Write the necessary code to publish an event */
  /*********************************************/
  /*********************************************/
  HelloMsg* msg=new HelloMsg;
  msg->data_="Hello from publisher";

  MessageDistributionSystem &msgSys=MessageDistributionSystem::getInstance();
  msgSys.notify(HELLO_MSG,msg);

  timer_->disArm(); // Timeout in TIMEOUT msec
  running_=false;
}
```

Følgende resultater er opnået ved brug af testen:

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe9/MDS$ ./main
2020-05-07 08:26:50.357 DBG (main.cpp:16 - main) Starting up...
2020-05-07 08:26:50.357 DBG (Publisher.cpp:12 - Publisher) Creating publisher with associated timer...
2020-05-07 08:26:50.357 DBG (Publisher.cpp:61 - run) Preparing for loop, arming timer...
2020-05-07 08:26:50.357 DBG (Subscriber.cpp:62 - run) Preparing for loop...
2020-05-07 08:26:50.357 DBG (Subscriber.cpp:62 - run) Preparing for loop...
2020-05-07 08:26:51.358 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
2020-05-07 08:26:51.358 DBG (Subscriber.cpp:29 - handleMsgIdHello) S(1) The hello message contained: 'Hello from publisher'
2020-05-07 08:26:51.358 DBG (Subscriber.cpp:29 - handleMsgIdHello) S(2) The hello message contained: 'Hello from publisher'
2020-05-07 08:26:52.359 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
2020-05-07 08:26:53.359 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
2020-05-07 08:26:54.359 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
2020-05-07 08:26:55.359 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
^C
```

Efterfølgende udgives programmet således at de også kan unsubscribe igen, ved at udvide klassen *Publisher*

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe9/MDS$ ./main
2020-05-07 08:33:30.12 DBG (main.cpp:16 - main) Starting up...
2020-05-07 08:33:30.12 DBG (Publisher.cpp:12 - Publisher) Creating publisher with associated timer...
2020-05-07 08:33:30.12 DBG (Publisher.cpp:62 - run) Preparing for loop, arming timer...
2020-05-07 08:33:30.12 DBG (Subscriber.cpp:62 - run) Preparing for loop...
2020-05-07 08:33:30.12 DBG (Subscriber.cpp:62 - run) Preparing for loop...
2020-05-07 08:33:31.11 DBG (Publisher.cpp:24 - handleMsgIdTimeOut) Got timeout, publishing message and rearming...
2020-05-07 08:33:31.12 DBG (Publisher.cpp:74 - run) Ensuring timers are stopped...
2020-05-07 08:33:31.12 DBG (Publisher.cpp:77 - run) Thread terminating...
2020-05-07 08:33:31.12 DBG (Subscriber.cpp:29 - handleMsgIdHello) S(1) The hello message contained: 'Hello from publisher'
2020-05-07 08:33:31.12 DBG (Subscriber.cpp:72 - run) Thread terminating...
2020-05-07 08:33:31.12 DBG (Subscriber.cpp:29 - handleMsgIdHello) S(2) The hello message contained: 'Hello from publisher'
2020-05-07 08:33:31.12 DBG (Subscriber.cpp:72 - run) Thread terminating...
```

Som det kan ses lykkes det for begge at subscribe og unsubscribe igen.

Exercise 2.3 - Design considerations

Meningen med at implementere et MDS er, at få en lavere kobling mellem systemets instanser. Det vil gøre at klasserne eller programmet bliver nemmere at udvide og eller genbruge, fordi det kun er et samlet sted (i MDS) man vil skulle tilpasse programmet. Men der er kun så længe de fælles besked identiteter er blevet defineret.

Man kunne godt have valgt at bruge normale integeres istedet for strings som besked identiteter. Det vil dog blive en smule mere kaotisk, hvis man nu har rigtig mange besked identiteter, og integeres vil derfor ikke være særlig sigende omkring for den enkelte besked. Derfor vil det være nemmere for programmøren, hvis det er besked identiteten i sig selv, som fortæller hvad den kalder. Sandsynligeheden for at man vælger den forkerte formindskes forhåbentligt også.

Singleton
Der bruges singleton i implementeringen af message distribution systemet. Dette skyldes at der ønskes en

klasse der kan facilitere besked-transfererne i et publisher subscriber system. Der ønsked kun at have en MessageDistributionSystem, da dette kan håndtere et givent antal subscribere og publishere. Desuden er det vigtigt at disse benytter samme instans, da de ellers ikke kunne sende eller lytte til hinanden. For at implementere denne funktionalitet vælges altså Singleton design pattern. Et alternativ havde været et globalt objekt, som alle trådene havde adgang til. Dette kræver dog at alle trådene for objektet med som parameter, eller at objektet ligger i disses scope. Ved Singleton implementationen, skal alle brugere af MessageDistributionSystem, blot have kendskab til klassen, og kan så hente den ud vha. funktionen getInstance, når de skal bruge den.
MessageDistributionSystem har en privat constructor, og der kan derfor kun oprettes en instans i funktionen getInstance, der opretter et statisk objekt af MessageDistributionSystem. Funktionen returnerer derefter en reference til dette objekt.
Da det er en statisk variabel der oprettes, er dens levetid ikke begrænset af dets scope, men den eksisterer derimod i hele programmets levetid. Objeket oprettes altså første gang getInstance kaldes, og eksisterer derefter i resten af programmets levetid.
Dette har dog en enkelt ulempe ift. thread safety. Man kunne forestille sig at to tråde kaldte getInstance "samtidig", uden at getInstance var kaldt tidligere i programmet. Hvis den ene tråd så kaldte constructoren, og efterfølgende mistede CPU'en, ville den anden tråd også kalde constructoren, her ville der altså oprettes to objekter, og hvilket objekt ville fremtidige kald så tilgå? Dette kan løses vha. scoped lock.
Det kunne desuden give en problematik at alle der har kendskab til klassen, kan skrive sig op til at modtage/sende beskeder til de andre tråde. Dette kunne give noget rod hvis en tråd begyndte at sende uønskede beskeder under et label fra en anden publisher. Dette er dog en problematik som programmøren forhåbentlig kan overkomme. Derudover er det jo smart at trådene ikke behøver at vide noget om hinanden for at overføre beskeder. Da dette giver en lav kobling, og gør det let at skalere systemet. F.eks. kan information fra en publisher, e.g. en sensor, blot sendes ind i MDS, uden at tage hensyn til hvem der skal have informationen og hvornår. Senere kan der så designes moduler der behandler denne information, og disse kan let skrive sig op til at modtage information fra sensoren i MDS.

Publisher/Subscriber (Observer)
Ved brug af denne struktur, bliver det nemt og muligt at holde styr på hvem/hvilke (subscribers) der skal opdatere/modtage hvilke beskeder. Publisher underretter alle sine subscribers, og de henter derefter de nye meddeleser fra publisher. Strukturen er implementeret i *MessageDistributionSystem.hpp*, hvor funktionerne *subscribe(), unsubscribe()* og *notify()* giver lidt sig selv. Her er det en cointainer som er knyttet til hver message, og derved holder styr på hvem og hvor mange der subscriber til den message. Af samme grund ved publisher også hvem der skal have en notifikation, og derfor at de skal opdatere og herved pulle den nye message fra publisher.

Mediator
Mediator strukturen gør det muligt at sende nedarvede data fra *osapi::Message* til alle modtager. Specielt ved denne implementering er, at ”mediatoren” bliver samlingspunktet for observerne. Mediatoren ”holder styr på” observernes afhængigheder med hinanden og giver nye informationer fra den ene obserever videre til at den anden. De forskellige observers kommunikere derfor indirekte til hinanden via mediatoren.
I dette tilfælde er MDS vores mediator, da den står for kommunikationen. Ved brug af dette opnås lav kobling, og at subscriberne ikke behøver at kende til hinanden, hvilket er nice (Y)

Tak, for den her gang Søren! - Have fucking fun uden os 🙂

2_1_3_SubPub.png (77 KB) Magnus Nygaard Lund, 2020-05-07 10:42
2_1_3_Unsub.png (69,7 KB) Magnus Nygaard Lund, 2020-05-07 10:42