The Message Distribution System

Søren Hansen <shan@iha.dk>
V1.0

Introduction

In this exercise you will use the OS API to implement a message distribution system that is modeled after the GoF Observer Pattern and allows you to send any data inherited from `osapi::Message` you would like, to any recipient(s). This is a very powerful tool to have, e.g. for your 3rd semester project, so be sure to complete this exercise.

The beauty of the message distribution system is that the sender (publisher) of messages does not know (or care) who - if any - receives its message(s). The receiver (subscriber), on the other hand, does not know (or care) who sent the message. This is an example of how low coupling can be used to make a system extensible.

Prerequisites

You must have a working OSApi and thorough understanding of inter-thread communication via message queues

Goal

Upon successful completion of this exercise, you will:

- Have implemented a generic message system that can be used as the inter-thread communication framework in any system. An important aspect is that decouples the publisher and subscriber.
- Have acquired knowledge about the singleton pattern.

Exercise Introduction

In this exercise, we will construct a *Message Distribution System* to distribute messages from senders/emitters/posters to receivers/subscribers. Inspect the overall UML diagram below to get an idea of how the system is intended to work.

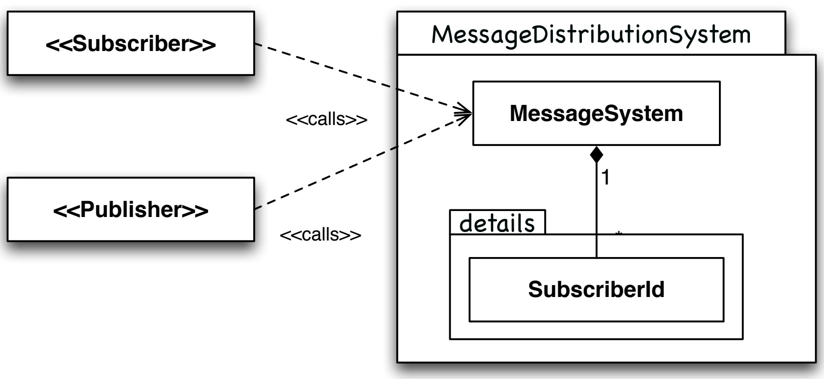

Figure 0.1: UML diagram showing relationships and usage

Søren Hansen <shan@iha.dk>

The Message Distribution System

V1.0

Before we "dive" into the exercise it is imperative that you consider which steps your code needs to perform. If you are uncertain where to begin, then completing it will most likely take a lot longer than strictly necessary.

Exercise 1 The Message Distribution System

The vision is to construct a system that is simplistic in its design as well as easy to use. Furthermore, it should also be easy to access, thus it should not be necessary to pass pointers around to get to use it. To facilitate this we will use the *singleton pattern* [Gam+95]. Below you will find the interface, note that the constructor is private, which is why we need a *static* function to construct the object. The static function contains a static variable which is constructed upon first use. The function returns a reference to the locally constructed object.

Why is the above approach legal? A reference to a local static variable returned from a function...

Listing 1.1: Class MessageDistributionSystem

```
namespace details
{
  /** Container for each subscriber, that holds relevant
   *  information that uniquely identifies a subscriber
   *  and ensures that when a message is send to the
   *  subscriber, it is done as per the subscriber requirements.
   */
  class SubscriberId
  {
  public:
    SubscriberId(osapi::MsgQueue* mq_, unsigned long id_);

    /** Send the message to the subscriber
     */
    void send(osapi::Message* m) const;

    /** Used to find subscribers in the vector
     */
    bool operator==(const SubscriberId& other) const
    {
      return (mq_ == other.mq_) && (id_ == other.id_);
    }
  private:
    osapi::MsgQueue*  mq_;
    unsigned long     id_;
  };
}

class MessageDistributionSystem : osapi::Notcopyable
{
public:
  /** Subscribes to the message that is globally unique and designated
   *  in msgId
   * msgId Globally unique message id, the one being subscribed to
```

```
   * mq    The message queue to receive a given message once one is
   *       send to the msgId.
   * id    The receiver chosen id for this particular message
   */
  void subscribe(const std::string& msgId,
                 osapi::MsgQueue* mq,
                 unsigned long id);

  /** Unsubscribes to the message that is globally unique and designated in msgId
   * msgId Globally unique message id, the one being subscribed to
   * mq    The message queue that received message designated by msgId.
   * id    The receiver chosen id for this particular message
   */
  void unSubscribe(const std::string& msgId,
                   osapi::MsgQueue* mq,
                   unsigned long id);

  /** All subscribers are notified
  /* whereby they receive the message designated with 'm' below.
   * msgId Globally unique message identifier
   * m     Message being send
   */
  template<typename M>
  void notify(const std::string& msgId, M* m) const
  {
    osapi::ScopedLock lock(m_);
    SubscriberIdMap::const_iterator iter = sm_.find(msgId);
    if(iter != sm_.end()) // Found entries
    {
      SubscriberIdContainer& subList = iter->second; // Why?

      for(SubscriberIdContainer::const_iterator iterSubs =
      subList.begin(); iterSubs != subList.end(); ++iterSubs)
      {
        M *tmp = new M(*m); // <-- This MUST be explained!
        iterSubs->send(tmp);
      }
    }
    delete m; // <- WHY? Could be more efficient implemented,
    // such that this de-allocation would be unnecessarily. Explain!
  }

  // Making it a singleton
  static MessageDistributionSystem& getInstance()
  {
    static MessageDistributionSystem mds;
    return mds;
  }


private:
```

```
  // Constructor is private
  MessageDistributionSystem() {}

  // Some form af key value pair, where value is signified by
  //  the msgId and the value is a list of subscribers

  typedef std::vector<details::SubscriberId>  SubscriberIdContainer;
  typedef std::map<std::string, SubscriberIdContainer> SubscriberIdMap;
  typedef std::pair<SubscriberIdMap::iterator, bool>   InsertResult;
  SubscriberIdMap          sm_;
  mutable osapi::Mutex     m_;
};
```

The overall idea for such a system is to keep track of which subscribers have subscribed to which messages. Therefore an associative container that can hold a list of subscribers and pair it with a message id string is needed. `std::map` is a sound choice, since it can do precisely that. The value part of this associative container must be another container[1], . The reason being that there might be more than one who desires to subscribe to a given message. The second container choice is a `std::vector`[2].

Exercise 1.1 Why a template function?

In the interface the `notify()` function is shown and implemented as a template function. Why is it imperative that it be a template function? Furthermore explain what the code does and how!

Hint: The first approach that comes to mind would be to use a `osapi::Message*` in the function signature of `notify()`, however this would break everything when trying to handle multiple receivers[3] - why?

Exercise 1.2 API Implementation

Before any tests can be performed the `MessageDistributionSystem` must be implemented, and currently four functions are missing their implementation.

- `SubscriberId(...)`
- `void SubscriberId::SubscriberId(...)`
- `void SubscriberId::send(...)`
- `void MessageDistributionSystem::unSubscribe(...);`

To get you going the following snippet is the implementation of for the function `void MessageDistributionSystem::subscribeMessage(...)`. It could be your first function in the file `MessageDistributionSystem.cpp`.

[1] Using a `std::multimap` *is* in fact an alternative

[2] Unless you know *exactly* what you are doing always use a vector in preference for any other sequential container (list is also a sequential container.)

[3] This is deliberately vague, but the different points lead to the answer

Listing 1.2: Implementation of function MessageDistributionSystem::subscribeMessage(...)

```
void MessageDistributionSystem::subscribeMessage(const std::string&
    msgId, osapi::MsgQueue* mq, unsigned long id)
{
  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);
}
```

To speed things up even further, a simple unfinished test harness has been created with the purpose of using the above class `MessageDistributionSystem`. You will find the unfinished test harness file `MessageDistributionSystemTest.cpp` in the same directory as you found this file.

For inspiration on how to implement the last unimplemented function in class `MessageDistributionSystem` mentioned above take a look at the template function `notify()` and the listing for function `MessageDistributionSystem::subscribeMessage(...)`. Finally do not start implementing *anything* before you have deduced exactly which steps are needed.

Exercise 2 RAII is important so lets use it here!

A very important problem with the solution so far that there is no guarantee that the subscription is unsubscribed when it is no more needed. In fact, if an active class completes its task and is deallocated, but in this process neglects to unsubscribe, then trouble is inevitable.
Why is this the case? and what really happens?.

To ensure that we have full control of our resources we employ the RAII idiom. Below is a listing that shows how such a RAII implementation could look like, however this is only the interface, the functions themselves have not been implemented. That is your job.

Listing 2.1: Class SubscriberHelper

```
class SubscriberHelper : osapi::Notcopyable
  {
  public:
    SubscriberHelper(const std::string& msgId,
                     osapi::MsgQueue* mq, unsigned long id);
    void unSubscribe();
    ~SubscriberHelper();

  private:
```

```
    const std::string& msgId;
    osapi::MsgQueue*  mq_;
    unsigned long     id_;
  };
```

To test and verify that your implementation works, use the test harness from the last exercise and modify it to use the above.

Exercise 3 Design considerations

Things to reflect about:

- What is the point of creating such a distribution system?
- Singleton
  - The design choice is a singleton, but what is the alternative and what would be the consequence?
  - When is `MessageDistributionSystem` created and when is it destroyed?
  - This particular implementation and its use has one particular drawback regarding thread-safety. When does this occur? How would you solve or ensure that this problem does not pose a significant problem?
  - A singleton is like a global variable... this means that all threads in an application have direct access to it and can subscribe or publish whatever they want... What do you think? - Good / Bad → elaborate!
- Do you foresee any particular problems by using simple global strings to designate a message?
- In exercise 2 the class `SubscriberHelper` contains the same data as the `MessageDistributionSystem` itself. Elaborate on who you would improve on this design.
- Which type publisher/subscriber scheme do you believe is in use here? Elaborate on why this is the case.
- Which type of decoupling can said about the `MsgQueue` and which type can be said about the `MessageDistributionSystem`? Why?