The Message Distribution System

Søren Hansen <sha@ase.au.dk>
V1.5

Introduction

In this exercise you will use the OS API to implement a message distribution system that is modeled using the Publisher/Subscriber, Mediator and Singleton Pattern and allows you to send any data inherited from `osapi::Message` you would like, to any recipient(s).

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 need to know (and might not even 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

---

Exercise 1 The Message Distribution System

Exercise 1.1 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.

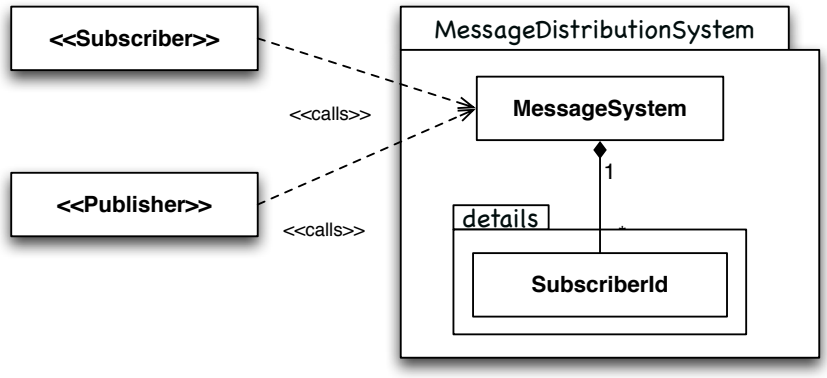

Figure 1.1: UML diagram showing relationships and usage

To get you started a complete harness has been pre-created. In the same dir you found this file you will find the `MDS.zip` file. Do yourself a favor and get familiarized with the contents. It compiles and will run out of the box.

Do note that the makefile and the header files are complete. You only have to make changes in the cpp files.

Exercise 1.2 Design and implementation

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*. 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: Excerpt from code/MessageDistributionSystem.hpp

```
1  class MessageDistributionSystem : osapi::Notcopyable
2  {
3  public:
4    /** Subscribes to the message that is globally unique and designated
5     *  in msgId
6     * msgId Globally unique message id, the one being subscribed to
7     * mq    The message queue to receive a given message once one is
8     *       send to the msgId.
9     * id    The receiver chosen id for this particular message
10    */
11   void subscribe(const std::string& msgId,
12                  osapi::MsgQueue* mq,
13                  unsigned long id);
14
15   /** Unsubscribes to the message that is globally unique and designated in
         msgId
16    * msgId Globally unique message id, the one being subscribed to
17    * mq    The message queue that received message designated by msgId.
18    * id    The receiver chosen id for this particular message
19    */
20   void unSubscribe(const std::string& msgId,
21                    osapi::MsgQueue* mq,
22                    unsigned long id);
23
24   /** All subscribers are notified
25    * whereby they receive the message designated with 'm' below.
26    * msgId Globally unique message identifier
27    * m     Message being send
28    */
29   template<typename M>
30   void notify(const std::string& msgId, M* m) const
31   {
32     osapi::ScopedLock lock(m_);
33     SubscriberIdMap::const_iterator iter = sm_.find(msgId);
34     if(iter != sm_.end()) // Found entries
35     {
36       const SubscriberIdContainer& subList = iter->second; // Why?
37
38       for(SubscriberIdContainer::const_iterator iterSubs =
39       subList.begin(); iterSubs != subList.end(); ++iterSubs)
40       {
```

```
41          M *tmp = new M(*m); // <-- This MUST be explained!
42          iterSubs->send(tmp);
43        }
44      }
45      delete m; // <- WHY? Could be more efficient implemented,
46      // such that this de-allocation would be unnecessarily. Explain!
47    }
48
49    // Making it a singleton
50    static MessageDistributionSystem& getInstance()
51    {
52      static MessageDistributionSystem mds;
53      return mds;
54    }
55
56
57  private:
58    // Constructor is private
59    MessageDistributionSystem() {}
60
61    // Some form af key value pair, where value is signified by
62    //  the msgId and the value is a list of subscribers
63
64    typedef std::vector<details::SubscriberId>  SubscriberIdContainer;
65    typedef std::map<std::string, SubscriberIdContainer> SubscriberIdMap;
66    typedef std::pair<SubscriberIdMap::iterator, bool>   InsertResult;
67    SubscriberIdMap         sm_;
68    mutable osapi::Mutex    m_;
69  };
70
71
72  #endif
```

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 1.2.1 Why a template method?

In the interface the `notify()` method is shown and implemented as a template method. Why is it imperative that it be a template method? 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 method signature of `notify()`, however this would break everything when trying to handle multiple different messages[3] - why?

[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

Exercise 1.2.2 API Implementation

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

- `SubscriberId(...)`
- `void SubscriberId::send(...)`
- `void MessageDistributionSystem::subscribe(...);` - Only one line of code is missing. Hint: What do you always need to consider when working with threads?
- `void MessageDistributionSystem::unSubscribe(...);`

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

Exercise 1.2.3 Test harness implementation

In the previous named MDS.zip file you also find a complete simple test setup that utilizes your now completed API. In the files `Subscriber.cpp` and `Publisher.cpp` some code is missing.

Fix it and verify that your solution works!

When you have acknowledge that it works try to unsubscribe in the receiving method in `class Publisher` upon receiving the first notification. Does your implementation still work?

Exercise 1.3 RAII is important so lets use it here (OPTIONAL)!

A very important problem with the solution so far, is 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. Have a look at the `SubscriberHelper class`, its header and implementation file. Work has commenced, but has not been completed. These methods are to be completed:

- `SubscriberHelper::SubscriberHelper(...)`
- `void SubscriberHelper::unSubscribe()`
- `SubscriberHelper::~SubscriberHelper()`

Questions to consider:

- How would I use this class?
- Why use a *destructive copy* in the assignment operation?
- How do we know that a subscription has been unsubscribed?

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

Exercise 1.4 Design considerations

Things to reflect about:

- What is the point of creating such a distribution system?
- Could I have used a simple integer instead of strings for a message identifier? Why use one over the other, what are the possible consequences?
- *Singleton*
  - We have chosen to use the *Singleton* design pattern, 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!
- *Publisher/Subscriber (Observer)*
  - What is the point of using this pattern in this context?
  - Where do I find it in the design/implementation?
  - What kind of mechanism are we using here? Push or Pull? And what is characteristic for these?
- *Mediator*
  - What is the point of using this pattern in this context?
  - How is it used?