Exam assignment summer 2023

Søren Hansen <sh@ece.au.dk>
V1.0

Introduction

This document describes the 2023 Summer exam for SW3ISU!

In these exam exercises you will use the OS API to implement the *Message Distribution System* and *PostOffice* design patterns[1]

The handin must contain a zip file with all source files, needed makefiles and a pdf with your answers!

Prerequisites

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

Exercise 1 The Message Distribution System (40%)

Exercise 1.1 Intro

In this exercise, we will finish constructing 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.

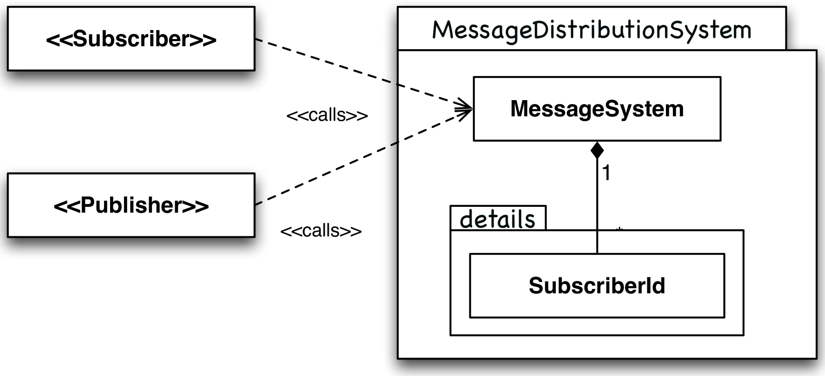

Figure 1.1: UML diagram showing relationships and usage

To get you started a complete suite has been pre-created. In the same directory you found this file you will find the `MDS.zip` file. Do yourself a favor and get familiarized with its contents. It compiles and will run out of the box. The implementation is almost complete, however, when instructed, you will be tasked to complete the missing parts!

[1]Do note that these are in no way design patterns on the same level as those explained in the GOF book.

Søren Hansen <sh@ece.au.dk>
V1.0

Exam assignment summer 2023

Exercise 1.2 Design and implementation

Exercise 1.2.1 Singleton

The goal 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*. In listing 1.1 the interface + implementation for this is shown.

Listing 1.1: Excerpt from `code/MessageDistributionSystem.hpp`

```cpp
static MessageDistributionSystem& getInstance()
{
  static MessageDistributionSystem mds;
  return mds;
}
```

Points to answer

- Explain how *Meyers* singleton works in C++.
  Note: Think about what a singleton is and how to achieve the desired effect is achieved in C++!
  Hint: What does the keyword `static` (has 3 meanings depending on placement) do and consider placement of the constructor and copy-assignment.
- Why is it actually valid code to return a reference to a locally constructed object?

Exercise 1.2.2 Types and variables

The overall idea for such a system is to keep track of which subscribers have subscribed to which messages.

Inspecting the code in listing 1.2 you will see the different types defined as well as the variables instantiated.

Listing 1.2: Excerpt from `code/MessageDistributionSystem.hpp`

```cpp
typedef std::vector<details::SubscriberId>  SubscriberIdContainer;
typedef std::map<std::string, SubscriberIdContainer> SubscriberIdMap;
typedef std::pair<SubscriberIdMap::iterator, bool>   InsertResult;
SubscriberIdMap          sm_;
```

Points to answer

- Explain what each of the types do and thus their responsiblity
- Which variables are instantiated and why?

Exercise 1.2.3 Why a template method?

In listing 1.3 the template method `notify()` is declared and defined.

Listing 1.3: Excerpt from `code/MessageDistributionSystem.hpp`

```cpp
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
  {
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

```
      const 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!
  }
```

This is implemented as a template method, which is somehow odd. One would have assumed that using `osapi::Message*` as an input type would have been sound, just as the `send()` method in the class `MsgQueue`.

Points to answer

- Why is it imperative that it is a template method?
- Explain what the code does and how!
  In your explaination ensure that you cover the two "whys" and the "...this must be explained..."
- Basics: *Who creates messages, who deletes them and how many are created and why?*
- If we were to use `std::shared_ptr<Message>` instead of the raw `Message*`, how would this affect the implementation?

Exercise 1.2.4 API Implementation

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

Tasks to complete

- Complete `SubscriberId(...)`
- Complete `void SubscriberId::send(...)`
- Complete `void MessageDistributionSystem::subscribe(...);`
  Only one line of code is missing. Hint: What do you always need to consider when working with threads?
- Complete `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 1.2.5 Test harness implementation

In the previous named *MDS.zip* file you also find a complete simple test setup that utilizes your now completed API.

Tasks to complete

- Complete the missing code parts in the files `Subscriber.cpp` and `Publisher.cpp`.

- Verify and document that it works
- Try to unsubscribe in the receiving method in `class Subscriber` upon receiving the first notification. Does your implementation still work?

Exercise 1.3 Design considerations

Points to answer:

- Could I have used a simple integer instead of strings for a global message identifier? Why use one over the other, what are the possible consequences?
- Inspecting this from a "resource" perspective. Do you foresee any challenges/problems?! IF so, could RAII be employed here and if so, where and how?

Exercise 2 The PostOffice design (35%)

Exercise 2.1 Intro

The *PostOffice* design is described in the associated presentation with all its simplicity and is thus considered the blueprint for its implementation for you in the following.

Remember that, just as for the previous exercise, this must utilize the OSApi.

Exercise 2.2 Task to do...

Exercise 2.2.1 PostOffice

Using the "blueprint" in the presentation create a class `PostOffice` having the API explained and what you find relevant. Explain What the class does, why it does it and how it is done. Use a sequence diagram.

Exercise 2.2.2 Design

Create a design having multiple threads that communicate using the *PostOffice*. Which design patterns are to be used to facilitate proper behavior?

The design should employ the *Timer* and the *logging* API in the OSApi to facilitate some timing example and reasonable logging of your choice.

Create a sequence diagram in which the relevant level of information is shown.

Exercise 2.2.3 Implementation

Implement your design making sure logging as well as proper makefiles for building your code is included.

Add relevant important snippets from your code implementation to document that you have implemented your design accordingly.

Exercise 2.2.4 Testing

Having designed, implemented and now executed your new program. Explain from its output that it actually does what is expected of it, as seen from the commandline.

Exercise 2.3 Design considerations

Points to answer:

- In what way does this change the design/implementation in your opinion?!
- There is a specific limitation to this design as presented for you in addressing other named threads.
  - What is the limitation
  - Exlain how/when this may be problem
  - Do you have an idea on how to solve this limitation?

Exercise 3 Overall design considerations (25%)

Points to answer:

- What is the point of creating such *Message Distribution System*?
- What is the point of creating such *Post Office*?
- How do the two different designs diverge?
- *Singleton*
  - The *Singleton* design pattern is employed in which of the designs and why? If the pattern were not to be used, what would be the alternative and associated perceived consequence?
  - Using *Meyers Singleton*, when is it created and when is it destroyed?
  - 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!
  - Regarding *Thread-safety*, when, why and where could/should I do something about threading?! Explain!
- *Publisher/Subscriber (Observer)*
  - In which design(s) do we see this design pattern employed?
  - Where/how do I find/see it in the design/implementation?
  - Which mechanism is employed here? Push or Pull? What is the characteristic for these?
- *Mediator*
  - In which design(s) do we see this design pattern employed?
  - How is it used?