Software Engineering
2016-17
Lab session 6
Subscriptions and notifications

Objective

1. Extend a simple system to a more complex one, keeping it under control.
2. Implement mechanisms for subscriptions and notifications.
3. Design methods and messages with a large degree of indirection.

Instructions

The design based on the Observer pattern from Seminar 7 will be implemented in two parts. This lab session represents the first part, and the goal is to implement several of the classes required for subscriptions and notifications. In Lab 7 the implementation will be refactored according to the design based on the Observer pattern.

In this lab session we are given a component that simulates the API (Application Programming Interface) of an outgoing mail server. This component is represented by the file `MailStub.hxx` that has been added to the `src/externalLibs` directory of your repository.

A stub or mock-up object is a fake component that substitutes a real component during tests. There may be several reasons for not including the real component during tests:

- because it is slow (remote calls, database access, etc.)
- because it is inconvenient (we do not want to send an email every time we execute the tests)
- because the system has not yet been implemented
- because we want our tests to be independent of the component

A stub shares the same interface with the real component and simulates its responses. Normally, a stub also offers an additional interface in order to verify that our component has interacted with the simulated component in the expected way.

`Singleton` pattern

Analyze the file `MailStub.hxx` and study

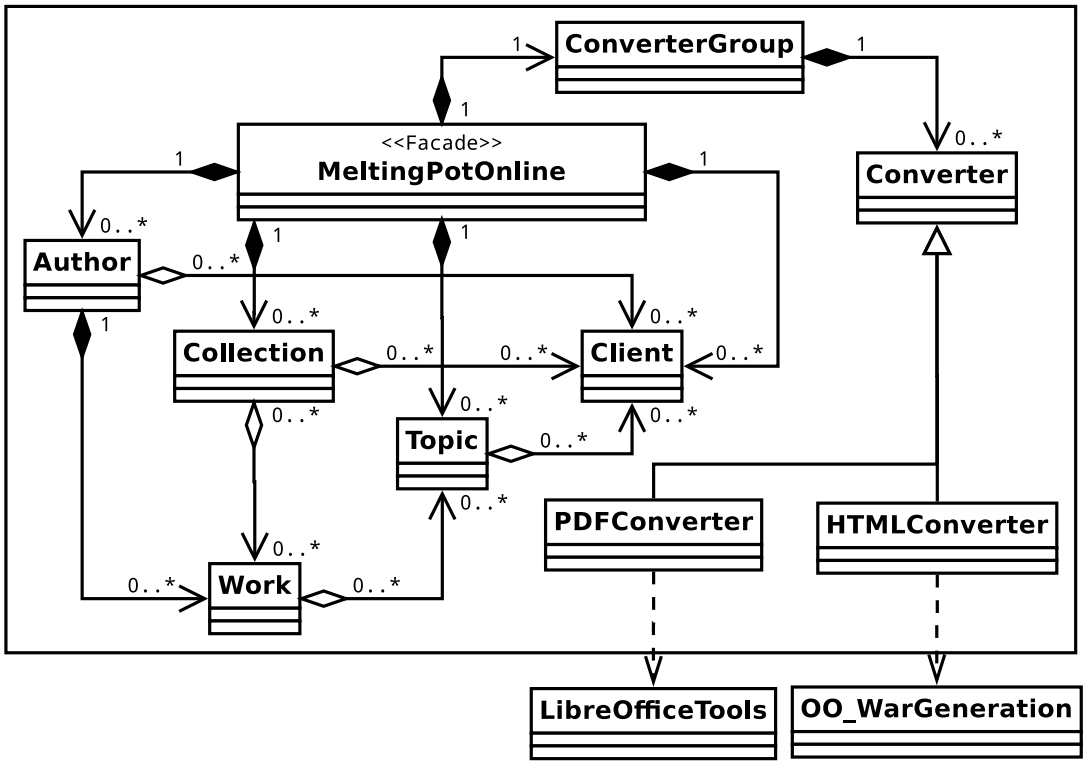
- how we can implement the `Singleton` design pattern in C++
- how we can extract a superclass that we can reuse every time we need to save a message log

Tasks

For each functional test in `SubscriptionTests.cxx`:

- Uncomment the test and perform a cycle RED-GREEN-REFACTOR
- If it is necessary to modify more than one class, introduce new unit tests and perform cycles RED-GREEN-REFACTOR

To organize the classes you may use the partial class diagram from the previous seminar:



We also provide a guide to passing the tests:

1. Implement the composition with topics in the class `MeltingPotOnline`.
    - Fail and pass the test `testListTopics_withOneTopic`. In the first version it is not necessary to create the class `Topic`.

- Fail and pass the test `testListTopics_withTwoTopics`. For the moment you can define an attribute of type
  `typedef std::vector<std::string>Topics;`

2. Implement the relation between works and topics:
   - Fail and pass the test `testAssociateTopicWithWork_withInexistentTopic`. To do so you have to define a new exception.
   - To pass the test `testAssociateTopicWithWork_workWithTwoTopics` you have to fail and pass a series of unit tests:
     - Create (using unit tests) a class `Topic` that includes a name attribute.
     - Refactor: Substitute the vector of strings in `MeltingPotOnline` by instances of the class `Topic`.
     - Test and implement, in the class `Work`, the multiple association with `Topic` (0,1,*) using a method `associateTopic` to establish a relation and a method `std::string topics() const` to test the relation.
     - Advice: Check the functional tests to see which format is required to return a list of topics in order to reuse the format in the method `catalogue` of `MeltingPotOnline`.
     - Test and implement, at the level of authors and works, that the catalogue description includes the topics of a work as expected in the functional tests.
     - Finally, test and create (using unit tests) the method `findTopic(topicName)` in the class `MeltingPotOnline`.
   - By now the test `testAssociateTopicWithWork_workWithTwoTopics` should be easy to implement.

3. Composition with clients:
   - Create the class `Client` with associated unit tests. The necessary attributes are inferred by the functional tests: name and email.
   - Fail and pass the test `testListClients_withOneClient` (a simple attribute in `MeltingPotOnline` is sufficient).
   - Fail and pass the test `testListClients_withTwoClients` (a collection is now required).

4. Subscribe clients to topics:
   - Start with `testListSubscribedToTopic_withOneUser`.
   - First you have to test a method `subscribeClient` in the class `Topic` using a getter `listSubscribed`.
   - Then you have to connect this method to `MeltingPotOnline` to pass the functional test.

- The other functional tests are relatively straightforward.

5. Finally, the functional tests involving notifications remain. These tests verify that when we associate a work with a topic, all clients subscribed to this topic receive a notification by email. It is evident that we have to use the method `associateTopicWithWork` of `MeltingPotOnline` to send the messages. We want the class `Topic` to notify clients since this class contains information about subscribed clients. To implement the notification mechanism we require the following tests:

   - Test and implement a method `update` of the class `Client`. This method has to receive the necessary information to compose the message. (Advice: pass the information in string format) The result (that should be tested) is that the singleton `MailStub` has a new message in its log.
   - Now we can move to the class `Topic` and test/implement a method `notify` with the same arguments as the method `update` of `Client`. The effect to be tested is that all messages have been sent to `MailStub`.
   - Finally, implement and pass the functional tests by delegating the work to the new methods implemented.

What to hand in?

The classes implemented in the `src` directory of your Git repository, and the unit tests in `src/unitTests`. You should also send an email to your lab session teacher in the format previously indicated.

When is the deadline?

Prior to the next lab session.