Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2011-2012

Homework 5

Due date: 28 May 2012, Monday, 23:55

1 Objective

This homework aims to help you get familiar with advanced object oriented concepts in C++. You will write a simple "Message Filtering" system similar to Mozilla Thunderbird's "Message Filters".

Keywords: *C++, OOP, Polymorphism*

2 Specifications

The system is composed of messages, filters and actions. For each message received, the system checks all filters and applies the action associated with the filter to the message. Following classes constitute the infrastructure of the system:

1. `Message`: Represents a message object with fields: "ID", "From", "To", "CC", "BCC", "Subject", "Body", "Date" and "Isread".
2. `MessageBox`: Represents a message box with sub-boxes, including but not limited to: "inbox", "sentbox", "drafts" and "trash".
3. `MessageClient`: Represents a client object that receives messages, supports filter registration and message search based on filters.
4. `Filter`: An abstract class that represents a message filter. You will derive from this class to create specific filters.
5. `Action`: An abstract class that represents an action that will be executed when a message passes a filter. Possible actions include "Copy to Box" and "Mark as Read".
6. `Exception`: Instances of this class will be used to specify exceptional cases.

2.1 Message

This is a really simple class that defines a message object. Messages IDs are guaranteed to be unique. “Isread” field will not be used for filtering.

```
#ifndef __MESSAGE_H__
#define __MESSAGE_H__

#include <string>
#include <iostream>

using namespace std;

class Message
{
public:
  /*----- DO NOT MODIFY/REMOVE  -----*/
  Message(const string& _id, const string& _from, const string& _to, const string↩
     & _cc, const string& _bcc,
      const string& _subject, const string& _body, const string& _date);
  const string& getID() const; // unique field, cannot be empty
  const string& getFrom() const; // non-empty string
  const string& getTo() const; // non-empty string
  const string& getCC() const; // can be empty
  const string& getBCC() const; // can be empty
  const string& getSubject() const; // can be empty
  const string& getBody() const; // can be empty
  const string& getDate() const; // non-empty string
  bool isRead() const; // not used for filtering
  void setIsRead(bool isread);
  /*----- DO NOT MODIFY/REMOVE  -----*/
  // TODO: add methods if required
private:
  // TODO: add required fields
};

#endif
```

2.2 MessageBox

This class keeps track of message boxes. It supports the default “inbox”, “sentbox”, “drafts” and “trash” boxes. In addition, it supports user-defined boxes. The ”putToBox” method creates a new box if it does not already exist.

```
#ifndef __MESSAGE_BOX_H__
#define __MESSAGE_BOX_H__
#include <string>
#include <vector>

#include "Message.h"

using namespace std;

class MessageBox
{
public:
  /*----- DO NOT MODIFY/REMOVE  -----*/
  MessageBox(); // used for possible initializations
  ~MessageBox(); // should remove all messages
  /**
   * Returns the messages that belong to box named "messageBox"
   */
  const vector<Message*>& getMessages(string messageBox) const;
  /**
   * Put the given Message in a box named "messageBox"
   * "messageBox" can be "inbox", "sentbox", "drafts", "trash" or
   * any other user-defined string identifier. If a box with the
   * given name does not exist, it will be created
   */
  void putToBox(Message* message, string messageBox);
  /*------ DO NOT MODIFY/REMOVE ------*/
  // TODO: add methods if required
private:
  // TODO: add fields/methods if required
};

#endif
```

2.3 MessageClient

This class is responsible for receiving messages, registering filters and performing filter-based message search. For each message, `MessageClient` checks all filters and executes the associated actions if required. All messages received via "receive" method goes to "inbox" after all filters are applied. Filters are applied in the order they are registered to the system. In case of conflicting actions, the last change will be reflected.

`MessageClient`'s `vector<Message*> search(const Filter* filter)` method returns all messages inside "inbox" that passes a given `Filter`. It should list the messages in the order they are received; i.e. you do not need to sort the messages.

```
#ifndef __MESSAGE_CLIENT_H__
#define __MESSAGE_CLIENT_H__
#include <vector>

#include "MessageBox.h"
#include "Filter.h"

using namespace std;

class MessageClient
{
public:
  /*------ DO NOT MODIFY/REMOVE ------*/
  MessageClient(); // used for possible initializations
  /**
   * Receives a message from the user, applies all
   * registered filters and saves the message to inbox.
   */
  void receive(Message* message);
  /**
   * Registers a Filter.
   */
  void addFilter(const Filter* filter);
  /**
```

```
   * Performs a filter-based search on messages in "inbox"
   * and returns those which pass the given Filter
   */
  vector<Message*> search(const Filter* filter);
  /**
   * Returns an instance of MessageBox class. This means
   * MessageClient holds an instance of MessageBox
   */
  MessageBox& getMessageBox();
  /*----- DO NOT MODIFY/REMOVE -----*/
  // TODO: add methods if required
private:
  // TODO: add fields/methods if required
};

#endif
```

2.4 Filter

This is perhaps the most important class of the system. It contains only pure virtual methods, i.e., it cannot be instantiated. You are expected to create subclasses of the `Filter` class to implement the required filters. Each filter contains a reference to an `Action` class, which will be executed if a message passes a filter. Note that `Filter` is not responsible for deleting the `Action` reference.

```
/* Filter.h
 *  Note: Do NOT modify this file!
 */
#ifndef __FILTER_H__
#define __FILTER_H__
#include <string>
#include "Message.h"
#include "Action.h"

using namespace std;

class Filter
{
public:
  /**
   * Gets/sets the name of the Filter
   */
  virtual string getName() const=0;
  virtual void setName(const string& name)=0;
  /**
   * This pure virtual method determines whether a message
   * passes this Filter or not
   */
  virtual bool pass(const Message* message) const=0;
  /**
   * Gets/sets the Action associated with this Filter
   */
  virtual const Action* getAction() const=0;
  virtual void setAction(Action* action)=0;
};
#endif
```

Here follows a list of filter classes you are supposed to implement. Each of them derives from the `Filter` class and will be placed in files named: "Filters.h" and "Filters.cpp'. Do not create a separate file for each filter.

1. `EqualsFilter(const string& field, const string& value, bool caseSensitive=true)`: This filter checks if the specified *field* of message is equal to the string *value*. For instance, `new EqualsFilter("↩ From", "support@example.com")` performs a case sensitive full-match against the "From" field of a message.

2. `ContainsFilter(const string& field, const string& value, bool caseSensitive=true)`: This filter checks if the specified *field* of message contains the string *value*. For instance, `new ContainsFilter("↩ Subject", "ceng140", false)` performs a case-insensitive partial-match against the "Subject" field of a message.

3. `BeginsWithFilter(const string& field, const string& value, bool caseSensitive=true)`: This filter checks if the specified *field* of message begins with the string *value*. For instance `new BeginsWithFilter↩ ("To", "e123456")` checks if the "To" field of a message starts with "e123456".

4. `EndsWithFilter(const string& field, const string& value, bool caseSensitive=true)`: This filter checks if the specified *field* of message ends with the string *value*. For instance, `new EndsWithFilter("↩ CC", "metu.edu.tr")` checks if the "CC" field of a message ends with "metu.edu,tr".

5. `DateFilter(const string& date, Operator op)`: This is used to filter messages according to their date. `Operator` is defined as: `enum Operator { EQUAL, LESS_THAN, GREATER_THAN }`. It compares the `date` argument with the date field of a message to decide if the message passes or not. If `Operator` is `LESS_THAN`, a `Message` passes this filter if its date is less than the argument `date`. Similarly, for `GREATER_THAN`, a `Message` passes this filter if its date is greater than the argument `date`. The format of a date is: `dd/mm/yyyy`. For instance, `05/09/2009`, `6/8/2012` or `4/4/2014` are valid dates.

6. `CompositeFilter(Composition comp)`: This filter is a composition of other filters. It allows filters to be added using the `CompositeFilter* addFilter(const Filter* filter)` method (notice that this method returns a reference to the object itself, i.e., return the "this" pointer, for syntactic purposes). `Composition` is an enum with the following definition: `enum Composition { ANY, ALL, NONE }`.

   - `ANY`: A message passes if at least one of the child filters passes.
   - `ALL`: A mesasge passes if all of the child filters pass.
   - `NONE`: A message passes if none of the child filters pass; i.e., if all child filters do not pass.

   Note that `CompositeFilter` is not responsible for deleting child filters when it is destructed. You may assume that there will be at least one child filter added to a `CompositeFilter`.

7. `NegateFilter(Filter* filter)`: This filter takes the negation of an existing filter. In other words, a message passes this filter if it does not pass the filter given as argument. For instance, `new ↩ NegateFilter(new EqualsFilter("BCC", "user@example.com"))` checks if the "BCC" field of a message is not equal to "user@example.com". Note that `NegateFilter` should not delete the wrapped child Filter when it is destructed. You may assume that the argument `Filter` of the constructor will not be `NULL`.

2.5 Action

This class represents an action to be executed if a message passes a filter. It contains only one method named "execute", which you need to override.

```cpp
/* Action.h
 *  Note: Do NOT modify this file!
 */
#ifndef __ACTION_H__
#define __ACTION_H__
#include "Message.h"
#include "MessageBox.h"

using namespace std;

class Action
{
public:
  /**
   * This pure virtual method takes a reference of MessageBox and Message.
   * MessageClient uses this method to execute the Action of a Filter
   */
  virtual void execute(MessageBox* messageBox, Message* msg)const=0;
};
#endif
```

You are expected to create two classes that inherit from the `Action` class. These classes will be placed in files named: "Actions.h" and "Actions.cpp". Do not create a separate file for each action.

- `CopyAction(const string& box)`: Copies a message to the box identified with *box*. A new *box* will be created if it does not exist.
- `MarkAsAction(bool isread)`: Sets the *IsRead* field of a message to *isread*.

2.6 Exception

This class is used to specify exceptional cases. There are two cases where you need to throw an instance of `Exception`:

- The `void addFilter(const Filter* filter)` method of `MessageClient` throws `Exception` if the action of a `Filter` is not set, i.e. if it is NULL. The code of the error is `ACTION_UNAVAILABLE`.
- The `const vector<Message*>& getMessages(string messageBox)const` method of `MessageBox` throws `Exception` if there are no messages associated with a given message box name. The code of the error is `BOX_NOT_EXISTS`.

Note: You are free to set error strings as you see fit; however, be careful to set the error codes correctly.

```cpp
/* Exception.h
 *  Note: Do NOT modify this file!
 */
#ifndef __EXCEPTION_H__
#define __EXCEPTION_H__
#include <string>

using namespace std;

class Exception
{
public:
  enum ErrorCode { BOX_NOT_EXISTS, ACTION_UNAVAILABLE };
  Exception(ErrorCode code, string msg) : errorCode(code), message(msg) {}
  void setCode(ErrorCode code) { errorCode = code; }
  void setMessage(string& msg) { message = msg; }
  int code() { return errorCode; }
  string what() { return message; }
private:
  ErrorCode errorCode;
  string message;
};
#endif
```

3 Sample Code

Here is a sample code that demostrates use-cases of the whole system. Beware that this code is far from a complete evaluation of the system. Its sole purpose is to illustrate the main structure of the system.

```
#include <iostream>
#include "Message.h"
#include "MessageBox.h"
#include "MessageClient.h"
#include "Filters.h"
#include "Actions.h"
#include "Exception.h"

using namespace std;

ostream& operator<<(ostream& os, const Message& message)
{
  // short output
  os << "Message::[ID:\"" << message.getID() << "\"][" << (message.isRead() ? "↩
     READ" : "UNREAD") << "]";
  // long output with headers
  /*
  os << "Message::[ID:\"" << message.getID() << "\"][From:\"" << message.getFrom↩
     () << "\"][To:\""
    << message.getTo() << "\"][CC:\"" << message.getCC() << "\"][BCC:\"" << ↩
       message.getBCC()
    << "\"][Subject:\"" << message.getSubject() << "\"][Body:\"" << message.↩
       getBody() << "\"][Date:\""
    << message.getDate() << "\"][" << (message.isRead() ? "READ" : "UNREAD") << ↩
       "]";
  */
  // multiple line output
  /*
  os << "From: " << message.getFrom() << "\nTo: " << message.getTo() << "\nCC: " ↩
      << message.getCC() << "\n"
    << "BCC: " << message.getBCC() << "\nSubject: " << message.getSubject() << "\↩
       n"
    << "Body: " << message.getBody() << "\nIsRead: " << (message.isRead() ? "Yes"↩
        : "No") << endl;
  */
  return os;
}


/**
```

```
 * Overloaded operator<< to print a Message vector to a stream
 */
ostream& operator<<(ostream& os, const vector<Message*>& items)
{
  vector<Message*>::const_iterator iter;
  for (iter=items.begin(); iter != items.end(); iter++) {
    Message* msg = *iter;
    os << *msg << endl;
  }
  return os;
}

/**
 * This methods prints a specific box of a MessageBox to a stream
 */
void printBox(ostream& os, const MessageBox& messageBox, const string& boxName)
{
  os << "--------------------------------------------------\n";
  try {
    const vector<Message*>& box = messageBox.getMessages(boxName);
    os << "[" << boxName << "]: " << "\n------------------------\n" << box;
  } catch (Exception& e) {
    os << "Error [" << e.code() << "]: \"" << e.what() << "\"" << endl;
  }
  os << "--------------------------------------------------\n";
}

/**
 * This function fills the argument vector with a number of Filter objects
 */
void getFilters(vector<Filter*>& filters)
{
  Filter* filter;
  Action* action;
  // Subject contains "Ceng140" (case-insensitive)
  filter = new ContainsFilter("Subject", "Ceng140", false);
  action = new CopyAction("ceng140");
  filter->setAction(action);
  filter->setName("ceng140");
  filters.push_back(filter);
  // From equals "erdal@ceng.metu.edu.tr" (case-sensitive)
  filter = new EqualsFilter("From", "erdal@ceng.metu.edu.tr");
  action = new CopyAction("erdal@ceng");
  filter->setAction(action);
  filter->setName("erdal@ceng");
  filters.push_back(filter);
  // CC not contains "ceng.metu.edu.tr" (case-sensitive)
  filter = new NegateFilter(new ContainsFilter("CC", "ceng.metu.edu.tr"));
  action = new CopyAction("notceng");
  filter->setAction(action);
  filter->setName("notceng");
  filters.push_back(filter);
  // Date after "10/10/2011"
  filter = new DateFilter("10/10/2011", DateFilter::GREATER_THAN);
  action = new CopyAction("after");
  filter->setAction(action);
  filter->setName("after");
  filters.push_back(filter);
  // Date before "8/9/2010"
  filter = new DateFilter("8/9/2010", DateFilter::LESS_THAN);
  action = new CopyAction("before");
  filter->setAction(action);
  filter->setName("before");
  filters.push_back(filter);
  // Body contains METU but not Hacettepe
  filter = (new CompositeFilter(CompositeFilter::ALL))->
    addFilter(new ContainsFilter("Body", "METU", false))->
    addFilter(new NegateFilter(new ContainsFilter("Body", "Hacettepe")));
  action = new MarkAsAction(true);
  filter->setAction(action);
  filter->setName("metu");
  filters.push_back(filter);
  // To, CC or BCC ends with "gmail.com"
  filter = (new CompositeFilter(CompositeFilter::ANY))->
    addFilter(new EndsWithFilter("To", "gmail.com"))->
    addFilter(new EndsWithFilter("CC", "gmail.com"))->
    addFilter(new EndsWithFilter("BCC", "gmail.com"));
  action = new CopyAction("gmail");
  filter->setAction(action);
  filter->setName("gmail");
  filters.push_back(filter);
  // To begins with either "erdal@" or "e144909", and
  // From not contains "gmail.com" and Subject contains "(No subject)"
  filter = (new CompositeFilter(CompositeFilter::ALL))->
    addFilter((new CompositeFilter(CompositeFilter::ANY))->
          addFilter(new BeginsWithFilter("To", "erdal@"))->
          addFilter(new BeginsWithFilter("To", "e144909")))->
    addFilter((new CompositeFilter(CompositeFilter::NONE))->
          addFilter(new ContainsFilter("From", "gmail.com"))->
          addFilter(new NegateFilter(new EqualsFilter("Subject", "(No subject)"))↩
             ));
  action = new CopyAction("complex");
  filter->setAction(action);
  filter->setName("complex");
  filters.push_back(filter);
  // Filter with a contradiction; it should not affect any messages
  filter = (new CompositeFilter(CompositeFilter::ALL))->
    addFilter(new EqualsFilter("From", "user"))->
    addFilter(new NegateFilter(new EqualsFilter("From", "user")));
  action = new CopyAction("nihil"); // box "nihil" will not be created
  filter->setAction(action);
  filter->setName("nihil");
  filters.push_back(filter);
  // Filter without an action; MessageClient should throw an Exception
  filter = new EqualsFilter("From", "user@example.com");
  filter->setName("no action");
  filters.push_back(filter);
}

/**
 * This function fills the argument vector with a number of Message objects
 */
void getMessages(vector<Message*>& messages)
{
  // ID From To CC
  // BCC Subject Body Date
  messages.push_back(
      new Message("1", "erdal@ceng.metu.edu.tr", "ceng140", "ceng140@ceng.metu.↩
         edu.tr",
        "", "Welcome to Ceng140", "Recitations start on February 27th. Please ↩
           fill your busy hour until.", "20/02/2012")
  );
  messages.push_back(
      new Message("2", "erdal@ceng.metu.edu.tr", "ceng242.news.ceng.metu.edu.tr",↩
          "erdal@ceng.metu.edu.tr",
        "erdal@metu.edu.tr", "ceng242 hw5", "Dear all; Hw5 is available on COW.",↩
           "14/05/2012")
  );
  messages.push_back(
      new Message("3", "erdal@ceng.metu.edu.tr", "ceng111.news.ceng.metu.edu.tr",↩
          "ceng111@ceng.metu.edu.tr",
        "", "Re: Welcome", "", "11/12/2012")
  );
  messages.push_back(
      new Message("4", "carlos@gmail.com", "chuck@gmail.com", "",
        "", "University Application", "Hi Chuck; You can apply to both METU and ↩
           Hacettepe for grad a position!", "05/04/2009")
  );
  messages.push_back(
      new Message("5", "e144909@metu.edu.tr", "erdal@metu.edu.tr", "",
        "", "(No subject)", "Nothing much to tell", "08/07/2008")
  );
  messages.push_back(
      new Message("6", "alice@crpyt.org", "bob@crypt.org", "",
        "carol@crypt.org", "Very secret message", "Benjamin Randell Harris was a ↩
           British infantryman who served in the British army during the ↩
           Napoleonic Wars.", "27/03/2012")
  );
}

int main()
{
  // create an instance of MessageClient
  //  all interaction will be done via this object
  MessageClient client;
  // get test filters and register them to MessageClient object
  vector<Filter*> filters;
  getFilters(filters);
  vector<Filter*>::iterator iter;
  for (iter=filters.begin(); iter != filters.end(); iter++) {
    try {
      // Register the Filter to MessageClient
      Filter* filter = *iter;
      client.addFilter(filter);
    } catch (Exception& e) {
      cout << "Error [" << e.code() << "]: \"" << e.what() << "\"" << endl;
    }
  }
  // create test messages and pass them to MessageClient object
  vector<Message*> messages;
  getMessages(messages);
  vector<Message*>::iterator it;
  for (it=messages.begin(); it != messages.end(); it++) {
    // let MessageClient handle each message
    Message* message = *it;
    client.receive(message);
  }
  // get the MessageBox reference from MessageClient
  MessageBox& messageBox = client.getMessageBox();
  // print default boxes (these exist even if they do not contain any messages)
  printBox(cout, messageBox, "inbox");
  printBox(cout, messageBox, "sentbox");
  printBox(cout, messageBox, "drafts");
  printBox(cout, messageBox, "trash");
  // print user-created boxes
  printBox(cout, messageBox, "ceng140");
  printBox(cout, messageBox, "erdal@ceng");
  printBox(cout, messageBox, "notceng");
  printBox(cout, messageBox, "gmail");
  printBox(cout, messageBox, "after");
  printBox(cout, messageBox, "before");
  printBox(cout, messageBox, "complex");
  // message boxes "nihil" and "test" do not exist
  printBox(cout, messageBox, "nihil");
  printBox(cout, messageBox, "test");
  //
  vector<Message*> filtered = client.search(new ContainsFilter("Body", "you", ↩
     false));
  cout << "[filtered]:" << endl;
  cout << filtered;
  return 0;
}
```

4 Regulations

1. **Programming Language:** You must code your program in C++. Your submission will be compiled with `g++` on department lab machines. You are expected make sure your code compiles successfully with `g++`.

2. **Late Submission:** At most 3 late days are allowed. After 3 days, you get 0. See the Syllabus for detailed information on our late policy.

3. **Cheating:** Both parties involved in cheating get 0 from all of the 6 homeworks and will be reported to the university's disciplinary actions committee.

4. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

5. **Grading:** This homework will be graded out of 100. It will make up 5% of your total grade.

5 Submission

Submission will be done via COW. Create a tar.gz file named `hw5.tar.gz` that contains all your source code files. This archive should not contain a source file with `main` function. Assuming that "sample.cpp" exists in the current directory, the following command sequence is expected to compile and run your program on deparment computers.

```
$ tar -xf hw5.tar.gz
$ g++ sample.cpp Message.cpp MessageBox.cpp MessageClient.cpp Filters.cpp Actions.cpp -o hw5
$ ./hw5
```