COMP 2404 -- Assignment #2

Due: Friday, February 14, 2014 at 11:00 AM

Goal

You will modify your movie database program from Assignment #1 and separate persistent objects into storage using the Façade design pattern. You will also implement a resizable collection class to hold the movie information.

Learning Objectives

- gain experience using a simple design pattern
- separate program logic for managing persistent objects
- practice with dynamically allocated memory in C++

Instructions:

1. Implement a resizable collection class

You will implement a dynamic array class (called **`DynArray`**) to hold the movies as a collection. The dynamic array should grow in size dynamically every time a movie is added and shrink in size every time a movie is deleted.

This class should:

- represent the movie collection as a dynamically allocated array of Movie pointers (i.e. a Movie double pointer)
- provide an add movie function to dynamically reallocate the movie collection and grow it by one element
- provide a delete movie function to dynamically reallocate the movie collection and shrink it by one element
- manage its memory to avoid memory leaks
- provide any additional utility functions that are required

2. Separate persistent data management

You will restructure your code so that the persistent data, i.e. the permanent ("master") movie collection, is exclusively managed by a separate **`Storage`** class. The **`Storage`** class will provide two member functions:

- **`void retrieve(DynArray* movieArr)`** which returns the entire contents of the movie collection
- **`void update(UpdateType action, DynArray* movieArr)`** which either adds to the permanent collection all the movies in the given movie array or deletes from the permanent collection all the movies in the given movie array. **`UpdateType`** is an enumerated data type that indicates either an add or a delete action.

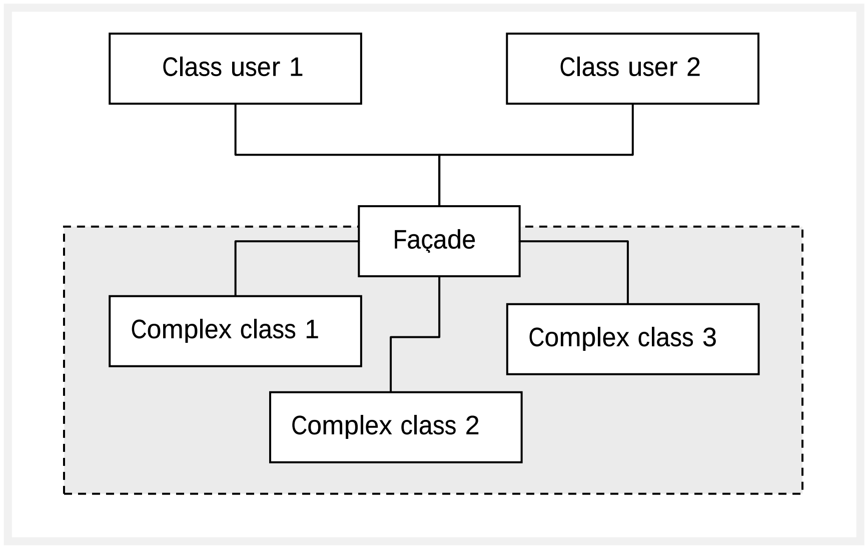
Notes:

- There can only be one instance of the **`Storage`** class in your program. Do not make copies.
- Only the **`Storage`** class is allowed to maintain the permanent movie collection, no other classes are allowed to.
- All classes in the program must send queries and updates to the **`Storage`** class when they need to display or modify movie information.

3. **Façade design pattern**

The Façade design pattern is commonly used to provide a simplified interface, like a gateway, to a set of complex classes. With this pattern, the class users do not need to understand the more complex operations of each class they need, but instead they can invoke a simple operation on the Façade class which itself uses the more complex operations on the needed classes.

Note: The Façade class does not *replace* the complex classes nor does it implement their functionality!



In a future assignment, you will be separating your program into a server process and a client process, where the server manages the persistent storage for multiple clients, while the client process contains the movie database logic and interacts with the user. Multiple clients will be able to interact with a single server, so that many users can share the same movie database. The Façade design pattern can be used on the client side as a special class that encapsulates the details of interacting with the server process.
For this assignment, you will begin the separation of client and server by implementing a Façade class, which you will call **`Server`**. All the classes in your program that perform queries and updates to storage will go through the **`Server`** class, which will pass along the queries and updates to the **`Storage`** class.

Notes:

- There can only be one instance of the **`Server`** class in your program. Do not make copies.
- Only the **`Server`** class is allowed to access the **`Storage`** class directly, no other classes are allowed to.

Constraints

- your program must be written in C++, and **not** in C
- do not use any functions from the C standard library (e.g. printf, scanf, fgets, sscanf, malloc, free, string functions)
- do not use any classes or containers from the C++ standard template library (STL)
- do not use low level data types when there is a better one available (e.g. do not use character arrays, use string objects instead)
- do **not** use any global variables or any global functions other than `main`
- do **not** use structs, use classes instead
- objects should always be passed by reference, not by value

Submission

You will submit in *cuLearn*, before the due date and time, the following:

- a UML class diagram (as a PDF file) that corresponds to your program design
- one **`tar`** file that includes:
  - all source and header files for your movie database program
  - a Makefile
  - a readme file that includes:
    - a preamble (program author, purpose, list of source/header/data files)
    - compilation instructions
    - launching and operating instructions

Grading

- **Marking breakdown:**

Component	Marks
UML class diagram	20
Resizable collection class	30
Storage class	25
Façade class	25

- **Deductions:**
  - Up to 50 marks for any of the following:
    - the code does not compile using `g++` in the VM provided for the course
    - the code cannot be tested because the program consistently crashes
    - the design does not generally follow correct design principles (e.g. data abstraction)
    - prohibited library classes or functions are used
  - Up to 20 marks for any of the following:
    - the Makefile or readme file is missing
    - global variables, global functions, or structs are used
    - objects are passed by value
    - the code is not correctly separated into header and source files
  - Up to 10 marks for missing comments or other bad style (non-standard indentation, etc.)
- **Bonus marks:**
  - Up to 5 extra marks are available for fun and creative additional features