CSE 241 Programming Assignment 6 (Double Point Assignment)

DUE

June 1, 2025, 23:55

Description

- This is an individual assignment. Please do not collaborate
- If you think that this document does not clearly describes the assignment, ask questions before its too late.

This assignment is about designing a simple class hierarchy for a given scenario in **C++**. **You have to use Abstract Classes(interfaces) and class inheritance.(extending classes). The aim of this assignment is to apply observer pattern. You have to use observer pattern.**

- Your program runs in terminal window and there won't be any file read write operations.

Elements of the problem

A class represents a dataset which stores objects. Some of the objects are visual objects such as video, image. Some of the objects are non-visual such as audio and text. Objects are also categorized into playable or non-playable. Text and image objects are non-playable. Video and audio objects are playable.

So, we have: - text: non-visual, non-playable - audio: non-visual, playable - video: visual, playable - image: visual, non-playable

Instances of these objects can be stored in the same dataset object. It may be good idea to start with a base class which can be stored in dataset. The you can derive (extend the base class) classes for text, audio, video and image.

Since we have classifications for these derived classes you can create Abstract classes(interfaces) for visual, playable, non-visual, non-playable. The derived classes can implement the related abstract classes.

In addition to these classes and abstract classes we have a class which represents a media player. And we have a class which represents a viewer. media player plays (this is just going to be a simulation) playable objects. viewer shows non-playable objects.

The simulation

You are going to create single dataset object. You will create various image, video, text and audio objects and add them to the single dataset. Then you will create multiple player and viewer objects which are going to be observers of the dataset. They will wait for changes in dataset for a particular category of objects. For example: if a new audio object is added to the dataset, the data set will send this information to the observers which are `player` type. `viewer` objects will not be notified.

You are going to populate the main function which extensively tests the classes you design.

Example:

```
//This is our dataset. We have only one dataset.
Dataset* ds = new Dataset();

//Lets create different observers.
Player* p1 = new Player();
Player* p2 = new Player();

Viewer* v1 = new Viewer();
Viewer* v2 = new Viewer();

//Lets register them to our dataset so that they can reach to data and updates.
//You can also register them to the dataset when you create them.
```

```
// ds has to figure out the type of the observer.
// it should not send unrelated information.
// For example, viewer objects don't want to know anything about playable objects.
ds->register(p1);
ds->register(p2);
ds->register(v1);
ds->register(v2);

//Dataset should also support un-registering.
//ds->remove_observer(p1);
//p1 no longer receives any update or list of items.



//Here we create different objects.
ds->add(new Image("imagename1", "dimension info1", "other info1"));
ds->add(new Image("imagename2", "dimension info2", "other info2"));
ds->add(new Image("imagename3", "dimension info3", "other info3"));
ds->add(new Image("imagename4", "dimension info4", "other info4"));
ds->add(new Image("imagename5", "dimension info5", "other info5"));

ds->add(new Audio("audioname1", "duration1", "other info1"));
ds->add(new Audio("audioname2", "duration2", "other info2"));
ds->add(new Audio("audioname3", "duration3", "other info3"));

ds->add(new Video("videoname1", "duration1", "other info1"));
ds->add(new Video("videoname2", "duration2", "other info2"));
ds->add(new Video("videoname3", "duration3", "other info3"));

ds->add(new Text("textname1", "other info1"));
ds->add(new Text("textname2", "other info2"));
ds->add(new Text("textname3", "other info3"));

//Lets use one of the player objects.

//We can get the currently playing object
Playable* po = p1->currently_playing();
//This prints info about the playing object.
po->info();
//we can remove this object from the dataset
ds->remove(po);
//po is no longer in the dataset. all of the interested observers will get this update.

//similarly, viewer object:
Non_playable* np = v1->currently_viewing();
np->info(); //this prints info about the object being viewed.



// player objects should support:
// show_list() --> shows the play list.
// currently_playing()  --> this returns the current object
// next(type)  --> plays the next object. does not return any object.
// if the type is "audio", it jumps to the next audio object.
// of the type is "video". it jumps to the next video object.
```

```
// previous(type) --> plays the previous object. does not return any object.
// Jumps according to the type info.
// player type observers just take a list of playable objects from ds and
// automatically plays the given list.
// If an item is removed from the dataset, same item is also removed from
// the playlist of every player type observer.
// Here we are not actually playing anything.
// You don't have to automatically skip to the next item.
// By default, currently playing is the first item in the received list.
// Unless we call next(), it is not going to change. If the list is empty,
// find a way to inform about this exception.
// If the currently played item is removed from the dataset, jump to the next.
// If there is no item, you can think this as an exception.
// You can either handle this by throwing an exception or
// you can use a "dummy" empty object which represents "no-item" case.
// (an empty list can store this non-removable "dummy" object)


//viewer objects should support:
//show_list() --> shows view list.
//currently_viewing() --> returns the current object.
// the rest is the same with player object.
// type info is either "text" or "image".
// next(type) and previous(type) jump according to the given type.


// in your tests, try to cover different cases.
// If an item is deleted, show that all the observers get the update for example.
```

Turn in:

- A Document(report) which includes UML description of your design. (50pts)
- Source code of a complete **C++** program and a suitable makefile.
- Provide comments unless you are not interested in partial credit. (If I cannot easily understand your design, you may loose points.)

- You cannot get full credit if your implementation contradicts with the statements in this document.