INF224 - Travaux Pratiques

Author

Castaneda Medina Laura Manuela

Overview

The project comprises two stages across different programming languages and frameworks, emphasizing object-oriented programming and GUI development:

1. *C++ Object-Oriented Programming*
2. *Java Swing Interface Development*

Together, these stages aim to teach comprehensive software development skills, from backend logic in C++ to frontend GUI development in Java Swing, culminating in a complete system for multimedia management and display.

Directory Structure

- `Castaneda_Medina__Laura_Manuela/`: Main project directory.
  - `cpp/`: Contains all C++ related files, including source code and Makefile.
  - `swing/`: Contains all Java Swing related files, including source code and Makefile.

C++ Component

This stage involves creating a multimedia set-top box software prototype, focusing on object-oriented design principles in C++. Base and derived classes where develop for multimedia objects (like photos, videos, and films), implement testing programs, and handle multimedia content effectively.

Compilation and Execution

Navigate to the `cpp/` directory and run `make run` to compile and execute the C++ program.

Java Swing Component

The second stage extends the project into Java, where a graphical user interface (GUI) was created using Java Swing. This GUI is designed to interact with the previously developed C++ software, allowing for multimedia object management and playback. This stage includes creating the main window with interactive elements, adding menu bars and toolbars, and establishing client-server communication for multimedia object retrieval and playback.

Compilation and Execution

Navigate to the `swing/` directory and run `make run` to compile and execute the Java Swing program.

Makefile

Both the C++ and Java Swing components include a Makefile for easy compilation and execution. Ensure you have the necessary compilers and JDK installed on your system.

Code Documentation

The code is documented using Doxygen. Refer to the generated documentation (*Documentation.pdf*) for a detailed overview of the codebase.

Questions C++

4e Etape: Photos et videos

1. The methods described, which are to be declared in the base class and overridden in the subclasses without implementation in the base class, are known as **pure virtual functions**. These functions are declared by assigning 0 to them in the declaration within the base class, making the base class abstract. This means objects of the base class cannot be instantiated directly, only objects of its subclasses can be.

   To declare this method:

   ```
   virtual void play() const = 0;
   ```

2. The reason why objects of the base class can no longer be instantiate after adding a pure virtual function is because the class becomes abstract. An abstract class is intended to provide a common interface and shared functionality for its subclasses, but it is incomplete on its own because of the presence of one or more pure virtual functions. Thus, we can only instantiate objects of classes that provide concrete implementations for all pure virtual functions inherited from their base class.

5e Etape: Traitement uniforme (en utilisant le polymorphisme)

1. The characteristic property of object-oriented programming that allows for the uniform treatment of a list containing both photos and videos, without concern for their specific types, is **polymorphism**. Polymorphism enables objects of different classes to be treated through the same interface, primarily through the use of base class pointers or references to access derived class objects. This capability is crucial for implementing a uniform behavior in a collection of objects that might belong to different classes but are related by inheritance.
2. In the case of C++, to make use of polymorphism, it is specifically necessary to:
   - Declare methods in the base class as virtual, so they can be overridden in derived classes. This includes both the methods for displaying attributes and for "playing" the objects.
   - Use pointers or references to base class types to refer to objects of the derived classes. This is because C++ needs to know at compile time the size of the objects it deals with, and using pointers or references allows the program to handle objects of different sizes (due to potentially different data members in derived classes).
3. The elements of the array in `main.cpp` should therefore be pointers (or smart pointers, like `std::unique_ptr` or `std::shared_ptr`) to the base class, not objects of the base class. This allows the array to hold references to both Photo and Video objects, treating them through their common base interface but allowing for the specific behavior of each derived class to be executed.
   - This is different from Java, where all non-primitive types are inherently handled via references, and you can directly store objects of derived classes in an array or collection of the base class type without the explicit need for pointers.
   - The reason for using pointers in C++ instead of objects is related to the slicing problem, where if objects of derived classes were assigned directly to elements of a base class array, any data or functions specific to the derived classes would be "sliced" off, leaving only the base class portion of the objects. Using pointers prevents slicing by allowing the program to access the full derived class objects.

7e étape. Destruction et copie des objets

1. **Which classes need modifications to prevent memory leaks when objects are destroyed?**
    - Any class that allocates dynamic memory with new must release that memory with delete to prevent memory leaks. The Film class allocates memory for an array of chapter durations, so it needs a destructor to deallocate that memory.
2. **What is the problem with copying objects that have instance variables which are pointers, and what are the solutions?**
    - The problem with copying such objects is the shallow copy behavior of the default copy constructor and assignment operator provided by C++. A shallow copy duplicates the pointer values but not the pointed-to data, leading to situations where multiple objects point to the same memory location. This can cause undefined behavior when one object modifies the data or deallocates the memory, affecting all objects sharing that pointer.
    - **Solutions include:**
        - Implementing a deep copy, where not only the pointers but also the pointed-to data are duplicated. This ensures each object manages its own copy of the data.
        - Using smart pointers (e.g., `std::unique_ptr`, `std::shared_ptr`) that automatically manage memory and can handle copying semantics correctly, though std::unique_ptr requires explicit handling for copy operations since it owns the resource exclusively.

8e étape. Créer des groupes

1. **Why use a list of object pointers?**
    - Using a list of pointers (specifically, pointers to the base class) allows the list to store objects of any derived class, enabling polymorphic behavior. This is necessary because the actual objects might have different sizes and behaviors, and we want to treat them through a common interface defined by the base class. This approach is similar to Java's use of references for all objects, where Java automatically handles objects polymorphically without needing to explicitly use pointers.
2. **Why objects are not destroyed when a group is destroyed?**
    - Since an object can belong to multiple groups, destroying an object when a group is destroyed would lead to undefined behavior when other groups try to access the destroyed object. This is managed by ensuring ownership of the objects' lifetimes is handled outside the group, typically by the code that instantiates the objects. This approach requires careful memory management to prevent memory leaks, as you must ensure objects are deleted when no longer needed.

10e étape. Gestion cohérente des données

1. **How to ensure objects are only created through the MediaManager class to maintain database integrity?**
    - Make the constructors of the multimedia and group classes private or protected, ensuring they cannot be directly instantiated with new by external code. Then, make MediaManager a friend class of these classes, allowing MediaManager to access their constructors while other classes cannot.

Questions Java

1ere Etape: Fenêtre principale et quelques interacteurs

1. When we launch the program and interact with it by clicking the buttons to add text and resizing the window:

1. **Text Addition:** Each button press adds a specific line of text to the `JTextArea`, as programmed. The two buttons add their respective text lines, demonstrating how button actions can manipulate other components in the UI.
2. **Window Resizing and Text Area Behavior:** Without a `JScrollPane`, the `JTextArea` might not handle overflow of text gracefully. As we add more text than the visible area can show, we might not be able to see all the text by scrolling. This is because `JTextArea` does not automatically include scrolling capability.
3. **The Need for JScrollPane:** To make the `JTextArea` fully usable, especially when the amount of text exceeds the visible area, incorporating it into a `JScrollPane` is necessary. The `JScrollPane` provides a scrollable view of the `JTextArea`, allowing to navigate through all the entered text regardless of the amount, addressing usability concerns that become apparent through interaction and window resizing.