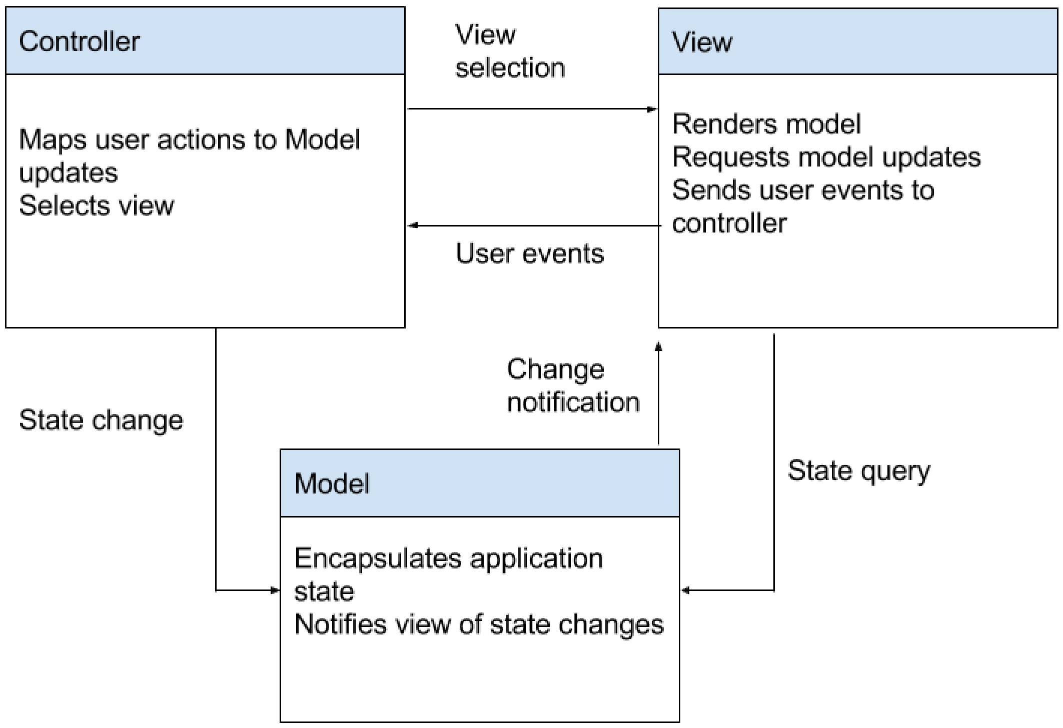
ECSE 321
Introduction to Software Engineering

Home Audio System
Deliverable 2
February 22 2016

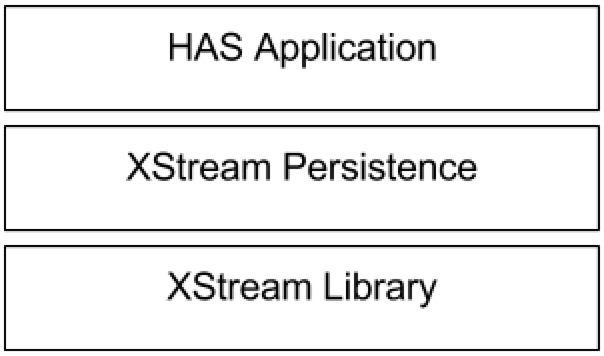
Kristina Pearkes 260 520 832
Aidan Piwowar 260 625 505
Kareem Halabi 260 616 162
Aurelie Pluche 260 622 575
Alexander Orzechowski 260 610 696

Architecture Block Diagram

Model View Controller



Layered



Detailed Architecture Description and Rationale

The above architectural block diagrams are the Model View Controller (MVC) architecture and the Layered architecture. We use the MVC architecture for Home Audio System (HAS) on each platform because it matches how data is handled through our application. This pattern allows multiple ways to view and interact with the data, enabling a simpler user experience. MVC also allows the freedom of changing the data independently of its representation which is done frequently in HAS. We encompass HAS in a Layered architecture because of how the application interacts with the XML file. The application requires the XStream persistence layer that uses the XStream library to read and write to an XML file.

The MVC architecture uses three key subsystems. The first being the model subsystem which handles the structure of the system data as well as the operations that can be used to manipulate it. Secondly, the view subsystem is used to create a user interface for which the user is able to submit and view data which, as mentioned previously, can be interchanged to allow multiple representations of data. Lastly, the controller subsystem is what manages interactions between the model and view, using operations from the model to deliver data back to the user through the view.

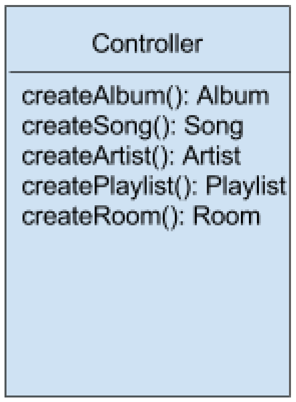
The Layered architecture is used for the XML persistence services in the application. This architecture organizes the system into layers that each add functionality and services to the layers above it. The top application layer communicates with XML Persistence requesting to save and load application data. Then, the XML Persistence layer organizes our application data so that it can communicate directly with the XStream library to load and save data to an XML file. The advantage of using this architecture for the HAS, is that if necessary we will be able to change or replace a layer while the interface is maintained. There is also an increase of dependability for the system due to redundancy between layers.

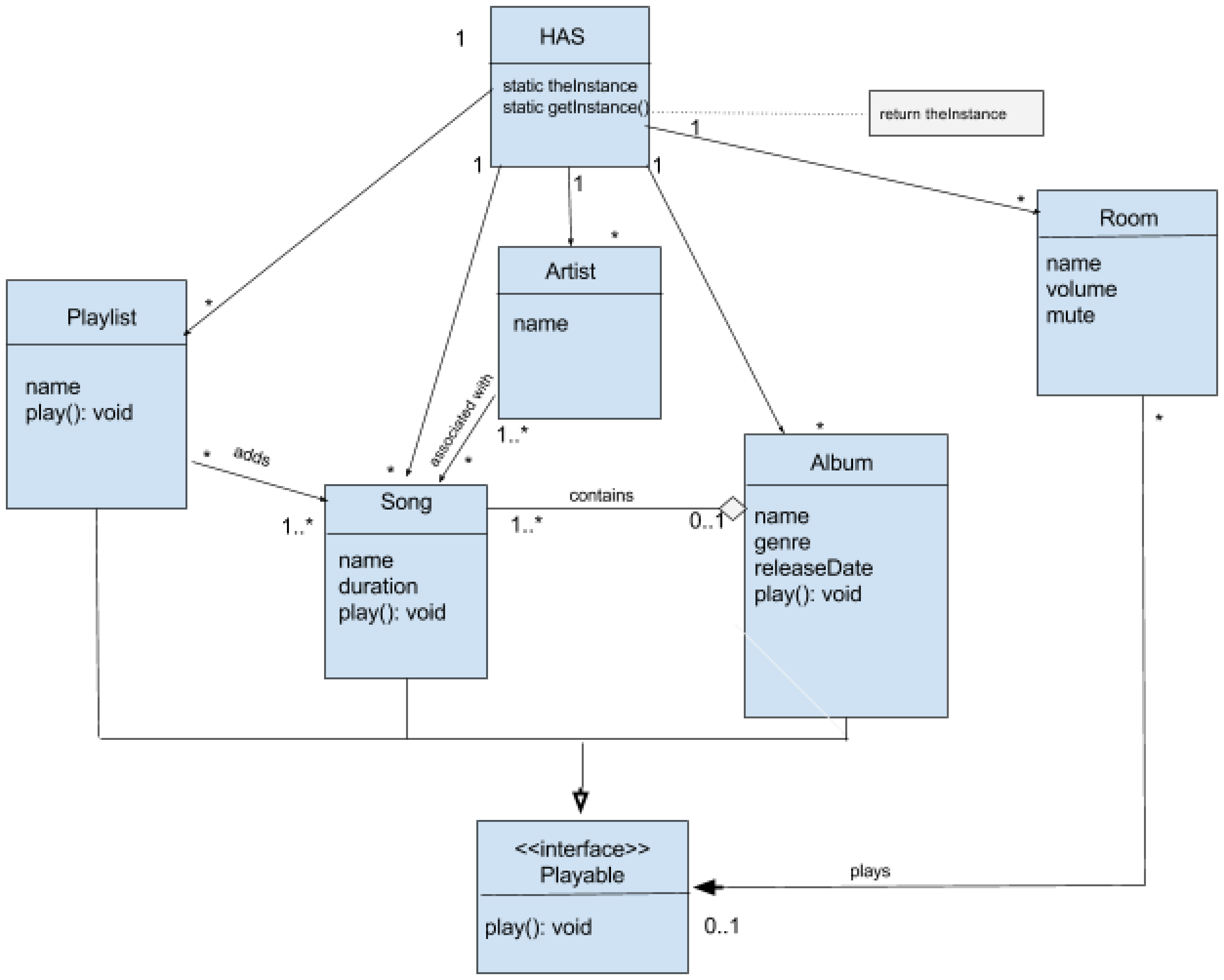
The "Add Album" functionality can demonstrate the communication between these subsystems. First, the view contains the user interface that allows the user to enter parameters. Then, the controller obtains these parameters to create a new Album in the model. If there is an exception, the controller will update the view to alert the user of the error. If the album is added successfully, the controller then makes a request to the XStream Persistence layer to save the model. XStream Persistence then organizes the data so that the XStream library can write the application data to an XML file. The view then updates itself to reflect the modified model.

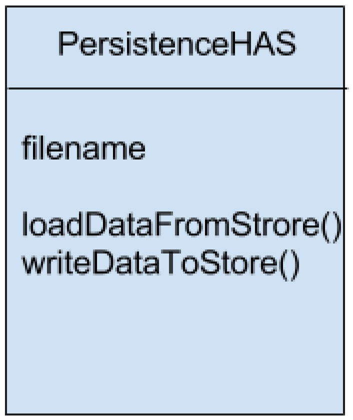
These two architectural styles are sufficient to capture all of the HAS requirements in full. The usage of another style is not only not necessary, but also may create some complications. These complications can render the concepts of the application vague and can create contradictions between the different architectures. The repository architecture would create discrepancies because of how the subsystems communicate with each other. As stated above, each subsystem communicates with one another, not through a single point as in the repository architecture. The use of a client-server architecture would be unnecessary since only one user will be using the same application state at one time. Lastly, the pipe and filter architecture is not used because the transfer of data is not sequential. This would disable the reusability of functions and transformations that manipulate the data.

The MVC and Layered architectures can be used for all three platforms: web, desktop and mobile. However, on a macro scale, PHP is a server-side scripting language with the web browser acting as a client and Xampp acting as the server. Therefore, on the web platform, the entire MVC & layered architecture is encompassed in a client-server architecture. The client-server behaviour is nothing that we have to maintain in the design of HAS, it is simply a consequence of using PHP.

Class Diagram of Web App







Detailed Design Description and Rationale

One design pattern we made use of is the singleton creational design pattern. We used a singleton root class called HAS, representing our Home Audio System object that can only have one instance. HAS knows about and manages all classes in the system. The reason we made HAS a singleton is because we want to make it accessible to people by using the code from a well-known access point. This results in strict control over how and when HAS is accessed which is a key part of the design of this Home Audio System because it makes the application easy to use and easy to organize data.

In our design we have also used the Strategy behavioural design pattern. We accomplish this with the Playable interface which lets the subclasses of Playable implement the play behaviour. We choose to use this design because playlists, songs, and albums can all be played, but they are played in different ways: a playlist and album implement the play behaviour by playing through a list of songs while a single song plays and then stops when the song is over. Moreover, the reason we choose to use the strategy pattern is because we want the algorithm that plays music to be selected at runtime.

Other than the HAS class, other key classes are the Room, Controller, Persistence and View classes. The Room class is responsible for playing music by accessing a Playable object which is either a Playlist, Album, or Song. Room can be thought of as the client code of our application because it accesses the music to be played and receives information from HAS that makes it able to send audio signals to any part of the house. Additionally, the Controller class is responsible for creating all objects when the user wants to add new music to their library. The Persistence class is responsible for loading and writing files to HAS when the user enters new music to be added and views existing music in HAS. Lastly, the view classes including “index.php” simply connect the view with the controller.

This class diagram of our application is not quite complete. The controller needs to be able to do more tasks such as add a song to a playlist, adjust the volume of music, and organize music by artist or album. These additions may change the controller design.

Work Plan

- Deliverable 1 Completed on February 22nd (High effort)
    - Add an album (2.1)
    - Add an artist (2.3)
    - Add a song to album (2.2)
    - Associate album to artist (3.1)
    - Functional and Non-Functional requirements
    - Domain Model
    - Use Case Diagram
    - Use Cases
    - Requirements-level sequence diagram for “Add Album” use case
    - Prototype of application on all three platforms for “Add Album” use case
- Deliverable 2 by March 7th (Medium effort):
    - Add a room (1.2)
    - Function in multiple rooms (1.3)
    - Create playlist(2.4)
    - Description of architecture of proposed solution including a block diagram
    - Description of detailed design of proposed solution including class diagram
- Deliverable 3 by March 21st (High effort):
    - Organize music by album or artist (3.2)
    - Choose a volume level for a room (1.4)
    - Be able to mute a room(1.5)
    - Select an item to play (1.1)
    - Description of unit testing
    - Description of component testing
    - Description of system testing
    - Description of performance/stress testing
- Deliverable 4 by March 28th (Low effort)
    - Description of release pipeline.
- Deliverable 5 by April 11/12/14 (Medium effort)
    - Presentation about project
- Deliverable 6 by April 15 (Medium-High effort)
    - Source code of full implementation on each supported platform