Lecture 18 : Build Spotify

Page No.
Date

Functional Requirements

- User can play and pause songs
- User can create playlist, add songs to playlist, ~~add songs~~ play entire playlist in any order
- App should support multiple output devices (Bluetooth speaker, wired speaker, headphones, etc.)

Non-functional Requirements

① Scalable
② New features / device / etc ~~can~~ should easily be added

Happy flow

- Application has multiple songs and playlists
- Each playlist contains different songs
- We can play songs using ~~more~~ playlist, pause that and play song according to our order.
- Application also supports different output device like Bluetooth speaker / Wired Speaker and headphones.
- Application should be scalable so we can add more devices.

Design Pattern Used -

Strategy Pattern, Singleton, Adapter, Facade, Factory

S1	→ Bluetooth Speaker
S2	→ Wired speaker
S3	→ Headphones
1 PL1 → S1, S2 } playAll()	
PL2 → S3, S4	

→ We make objects in application. Here we try to make managers of objects also so accessing an object functionality we have to contact its managers

Description and UML Diagram

→ First create a song class with attributes

Song
String name; String artist; String path; → assumed to already exist // methods

Functionality implementation

1. Play and Pause songs
- To make this work we treat output devices like Bluetooth speaker, wired speaker & headphones as third party APIs.

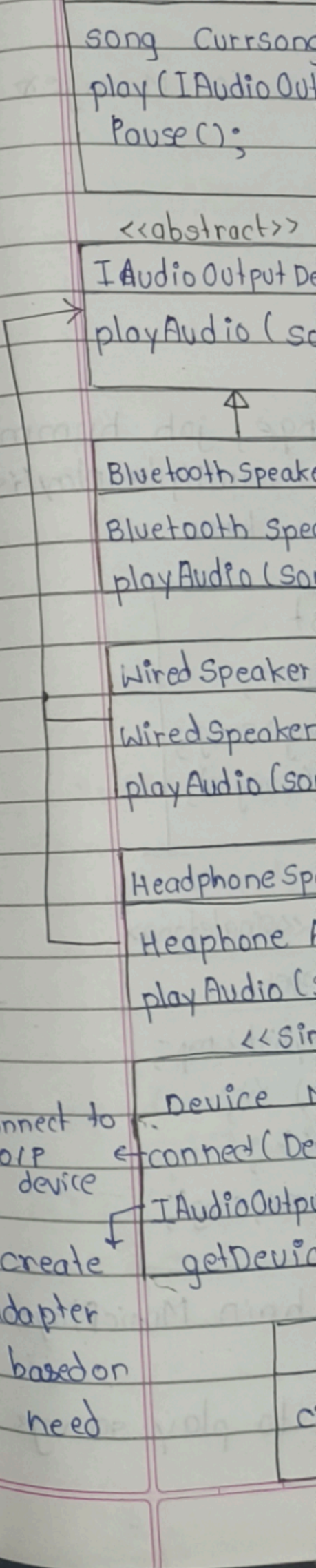
- To achieve this we will use Adapter Pattern to integrate third party APIs, which will make our design scalable.

- And now client only needs to play audio adapter will handle API call to respective 3rd party.

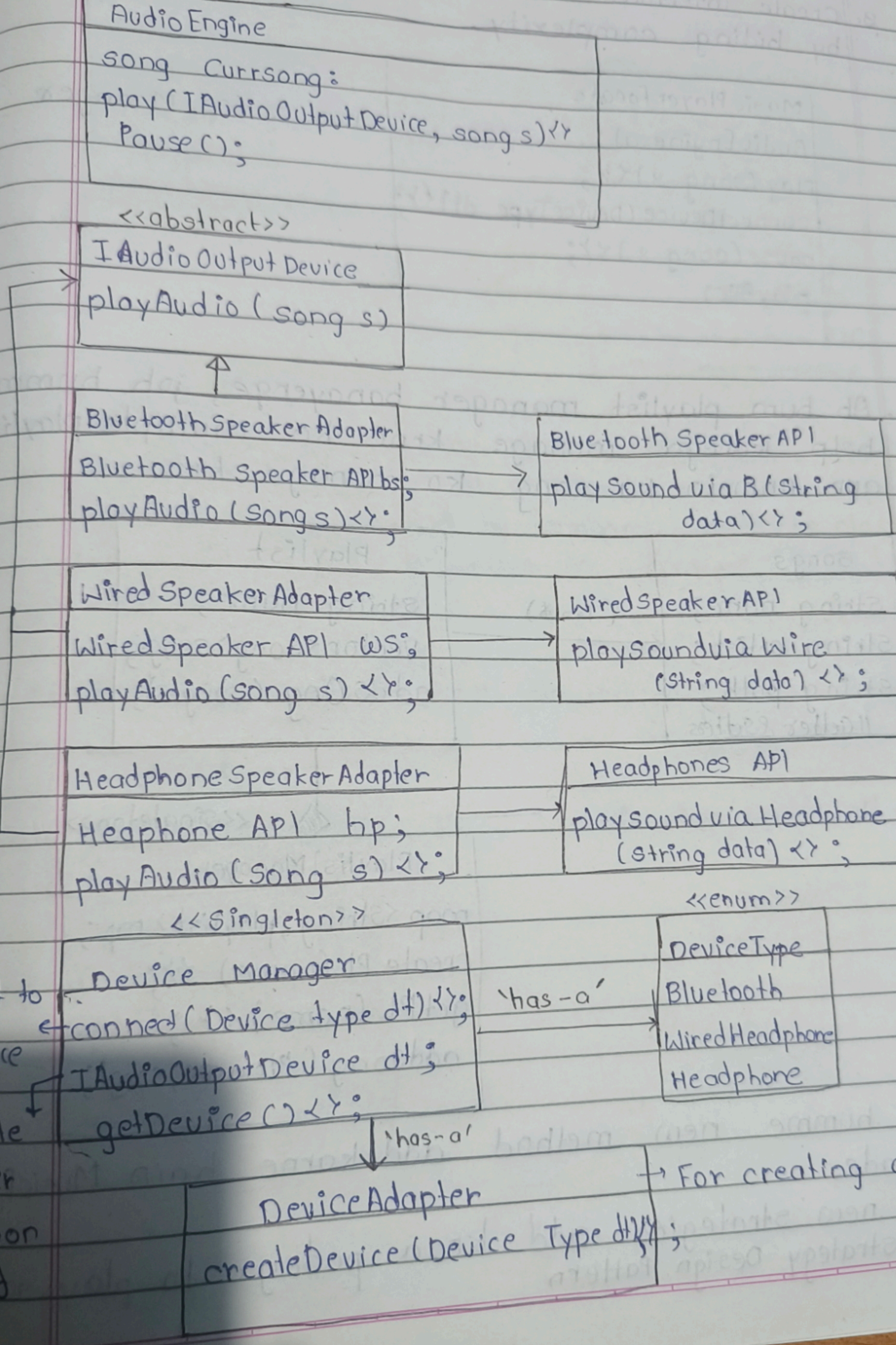
2. Audio Engine
- Ab hum jb playAudio() func. ko call krege toh internally konsa concrete class call hai and uske respective adaptee ko call karega.

- Pause stop krega current song ko.

- Aur ab De
saare adapt



- Aur ab Device Manager manage karega ~~all~~ saare adapters.



- IAudioOutputDevice ——▶ Device Manager
- DeviceFactory has a Bluetooth, wired, Headphone Adapter

Page No.
Date

3. Create interface for letting user interact with it by hiding complexity.

MusicPlayerFacade	has Device Manager
AudioEngine al; play(song s){}; connectDevice(DeviceType dt){}; pause(song s){}; playAll()	

4. Ab hum playlist manager banayenge, joh humme help karega manage krne ko multiple playlists aur unmein ka song ko manage krna.

songs
String name; String artist; String path; //Getter & Setters

(1..*)

Playlist
string name; vector<song> songs; addsOng(song s);

<<singleton>>

PlaylistManager
map<String, playlist> mp; createPL(name){}; getPL(name)<>; addsong(Plname, song);

→ Ab humme new method add karne hain MusicPlayer Facade.

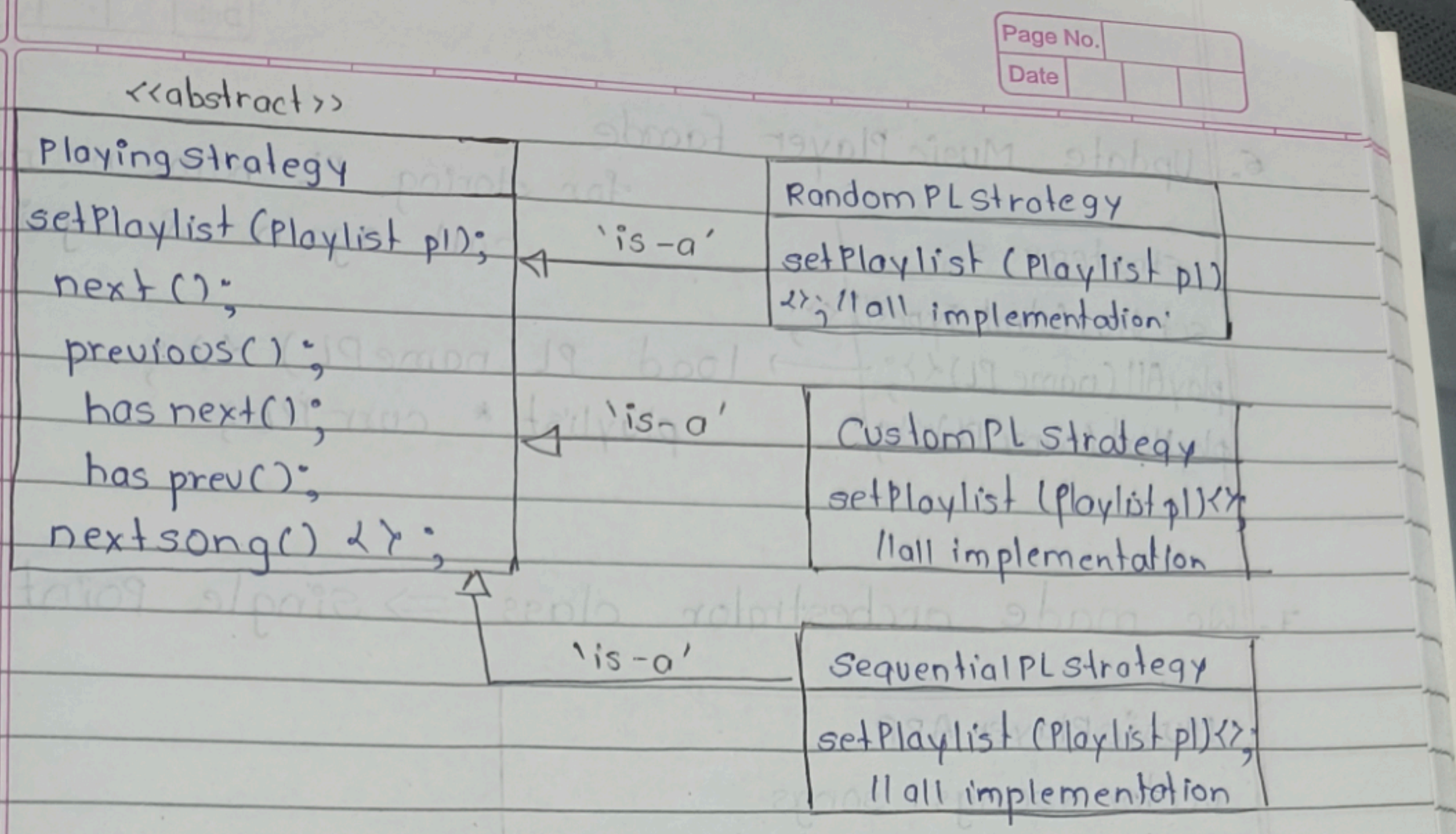
→ Ab new strategies add krna padega to play songs
⇒ Strategy Design Pattern

→ Ab yaha, we in abstract cl Random only we give en maintain I

<<abstract>>
PlayingStrategy
setPlaylist(Playlist pl
next();
previous();
hasnext();
hasprev();
nextsong()<>

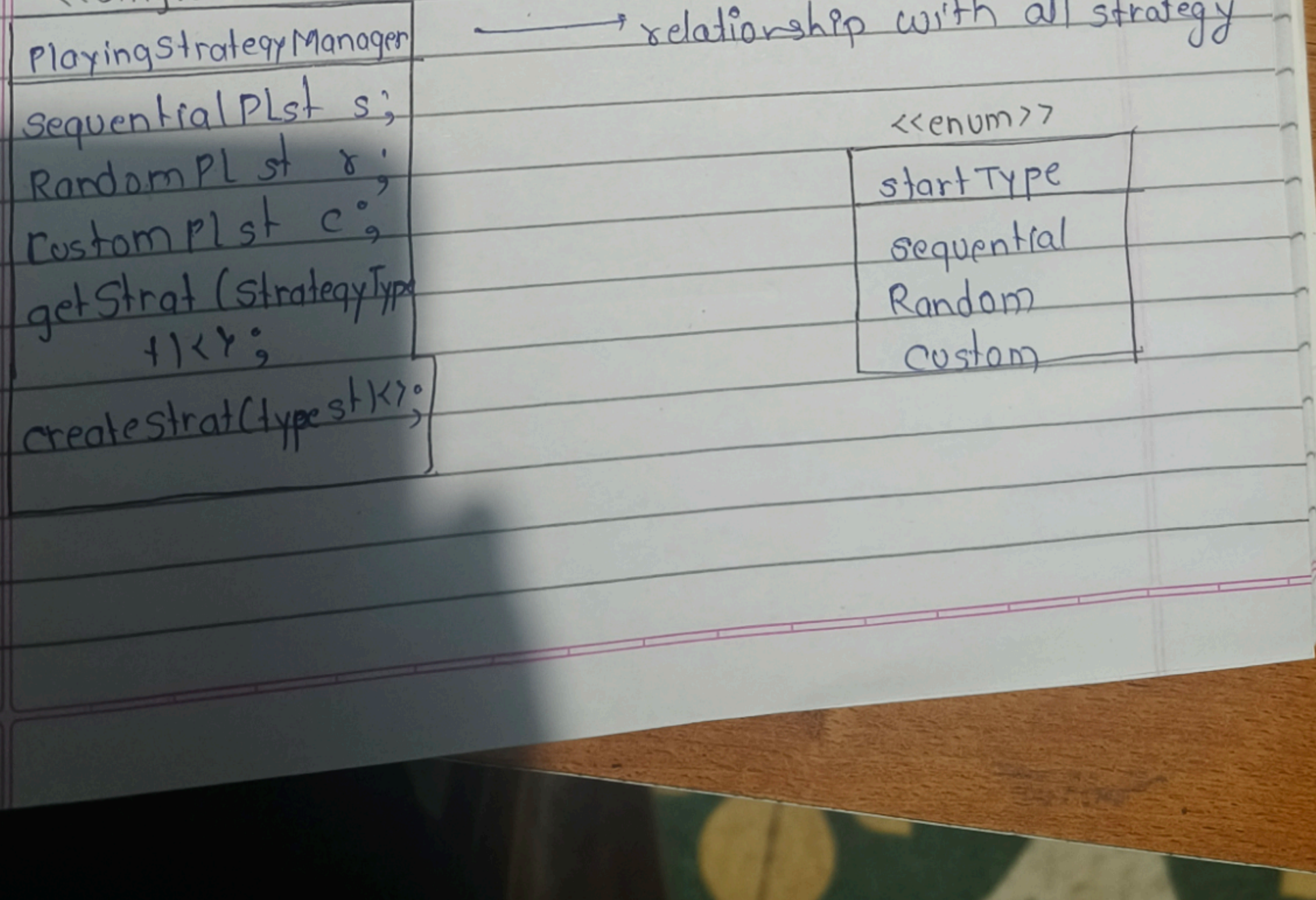
5. Maintaing St

<<singleton>>
PlayingStrateg
SequentialPL
RandomPL
CustomPLs
getStrat(s
createStrat



→ Ab yaha, we not only declare but define nextsong() in abstract class as it is not needed by sequential, Random only needed by custom concrete class. So we give empty defination in abstract class to maintain ISP

5. Maintaing Strategy ⟶ playing strategy Managen



6. Update Music Player Facade

for storing PL name no need to call again & again

strategy s;
setStrat(stratType'st)<>;
playAll(namePL)<>;
playNext(namePL)<>;

→ loadPL(namePL)<>;
playlist * currPL;

7. We made orchestrator class ⇒ single point Entry

MusicPlayerAPP
vector<songs> songs
createSong(name, artist, path);
createPL(name)<>;
playAll;
pbyNext;
playPrev;

} → contact facade