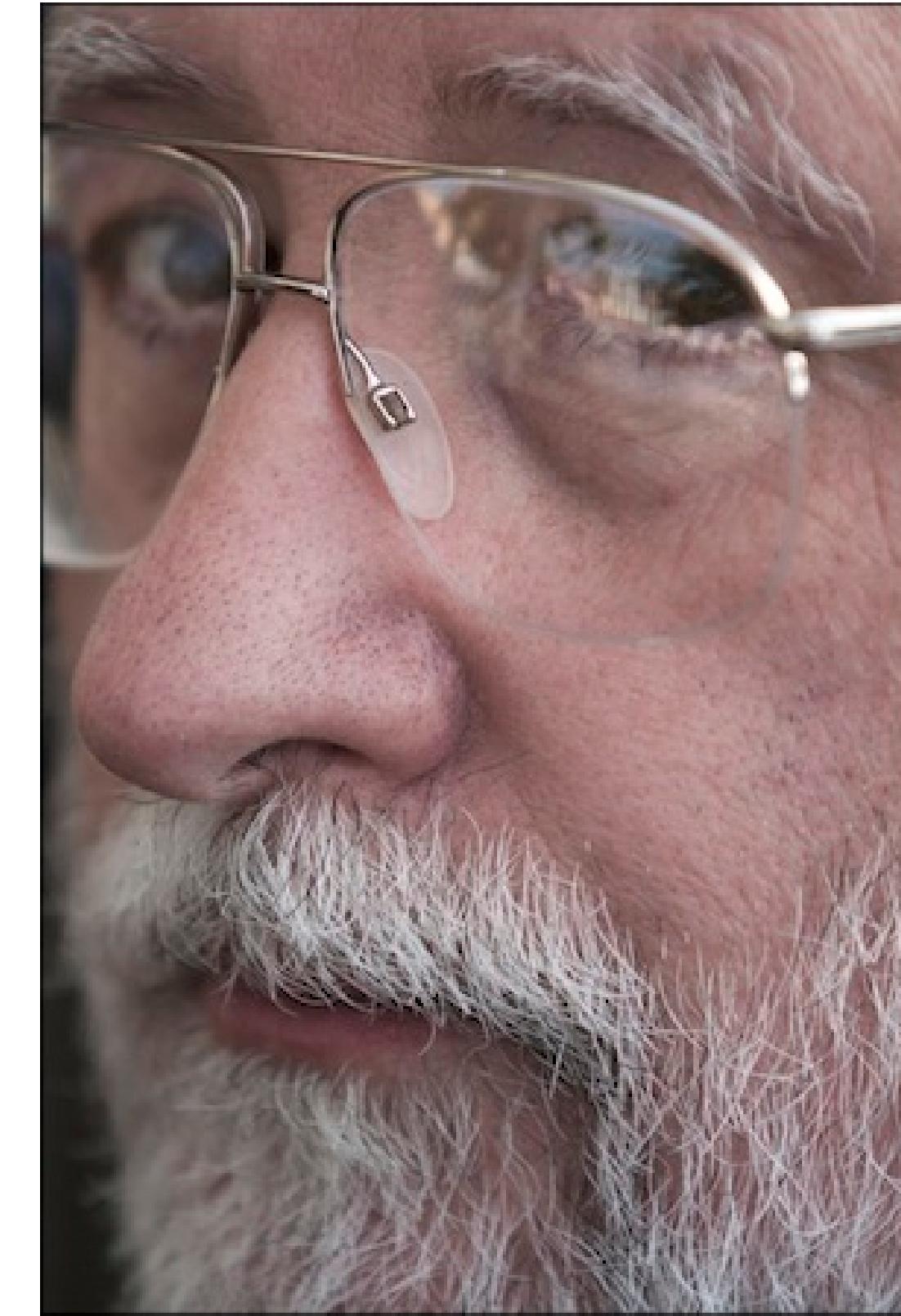


CS 319 DESIGN
PATTERNS AND
CODE SMELLS

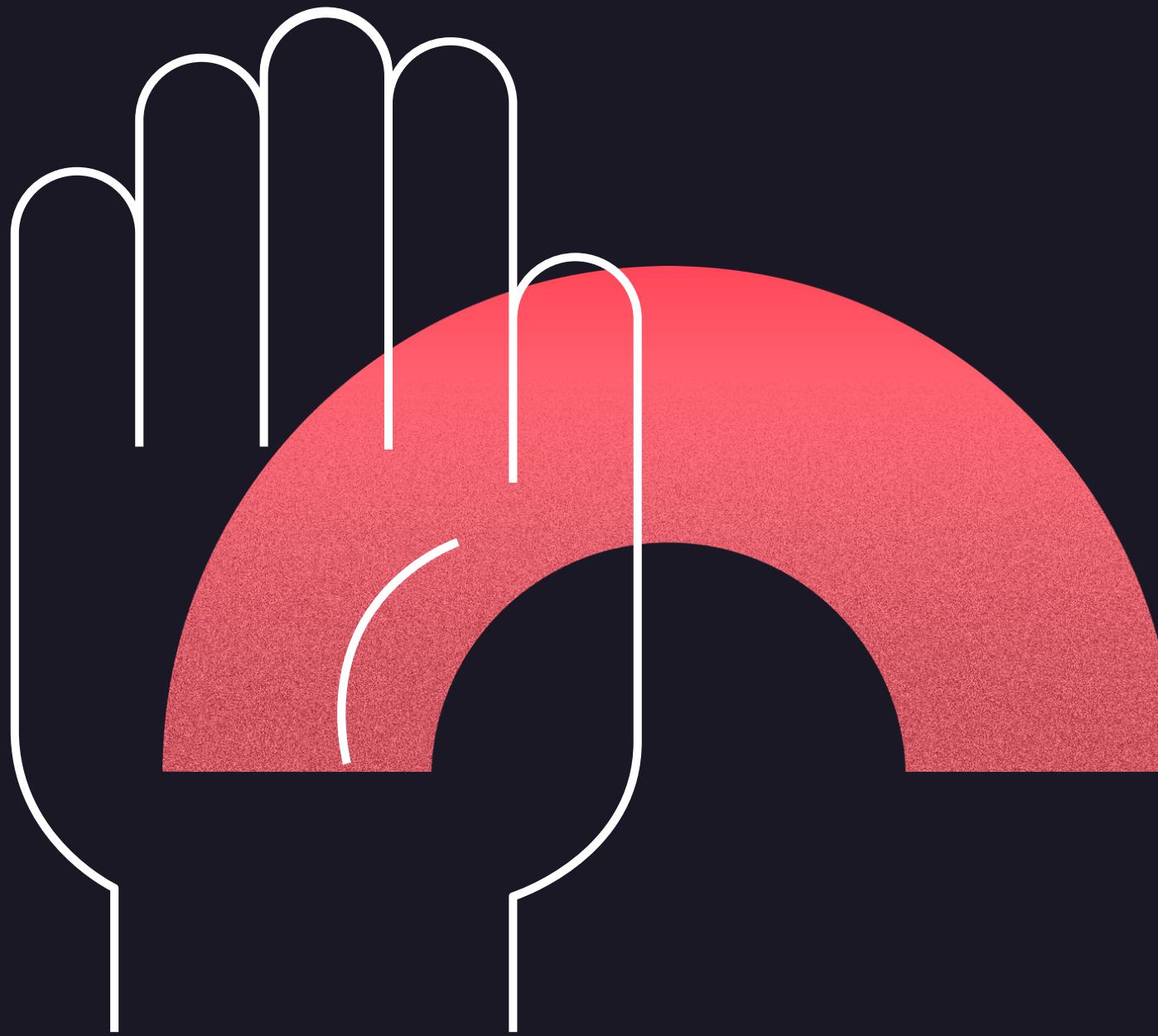
Mehmet Ata Yurtsever

GABRIEL IMPLIES THAT CODE SPEAKS TO THE PROGRAMMER THE SAME WAY A POEM SPEAKS TO THE POET. JUST AS ARTISTS LOOK FOR ASYMMETRY, IMBALANCE, OR LACK OF RHYTHM, DEVELOPERS LOOK FOR DEFICIENCIES IN THEIR CODE.

[1]



RICHARD P. GABRIEL

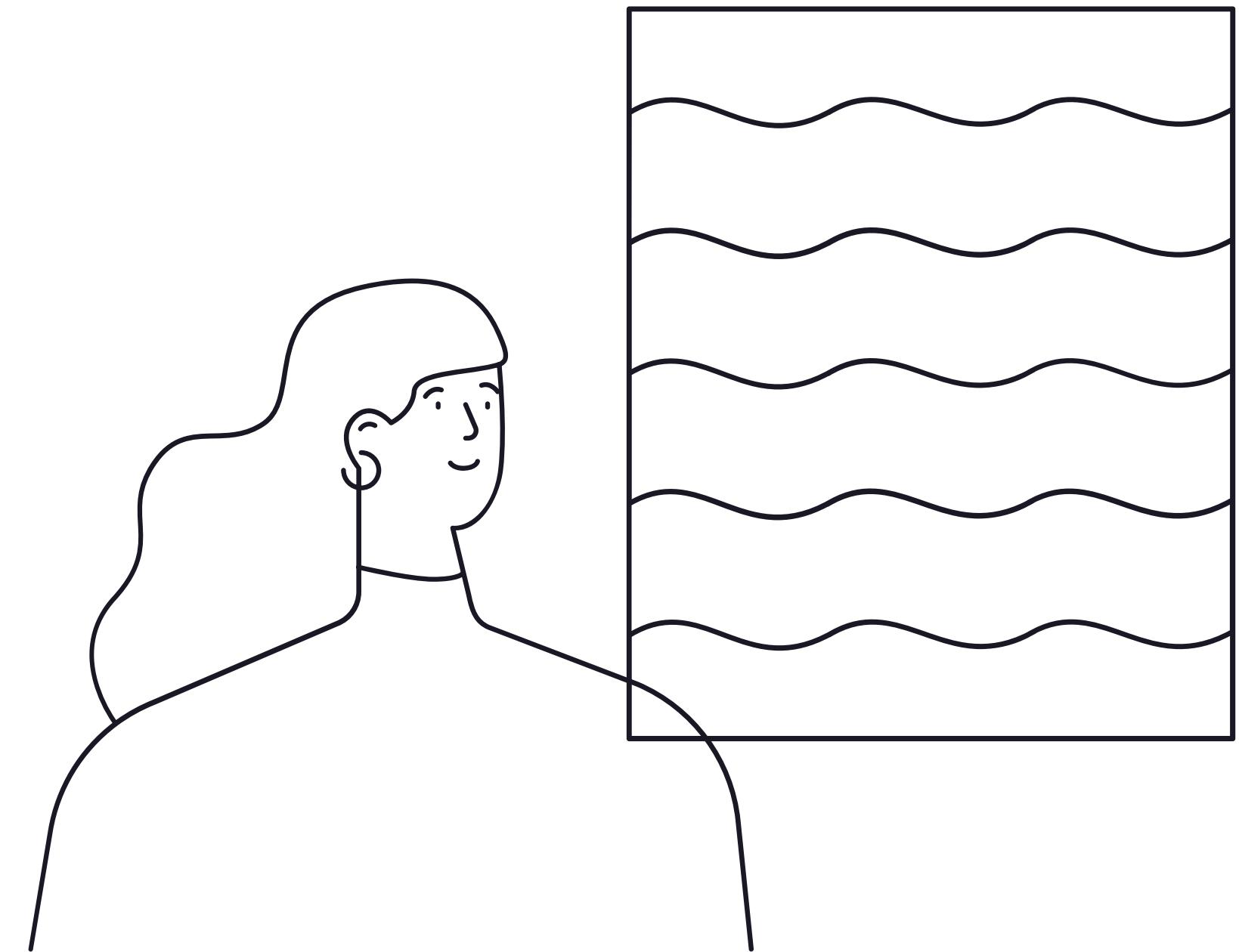


WHAT TO LISTEN FOR

If someone is good enough at “listening” to the thing they are designing, they will be guided by the artifact itself to its own natural design.

**Well, code
that is not
being changed
is not that
interesting to
listen to.**

[1]

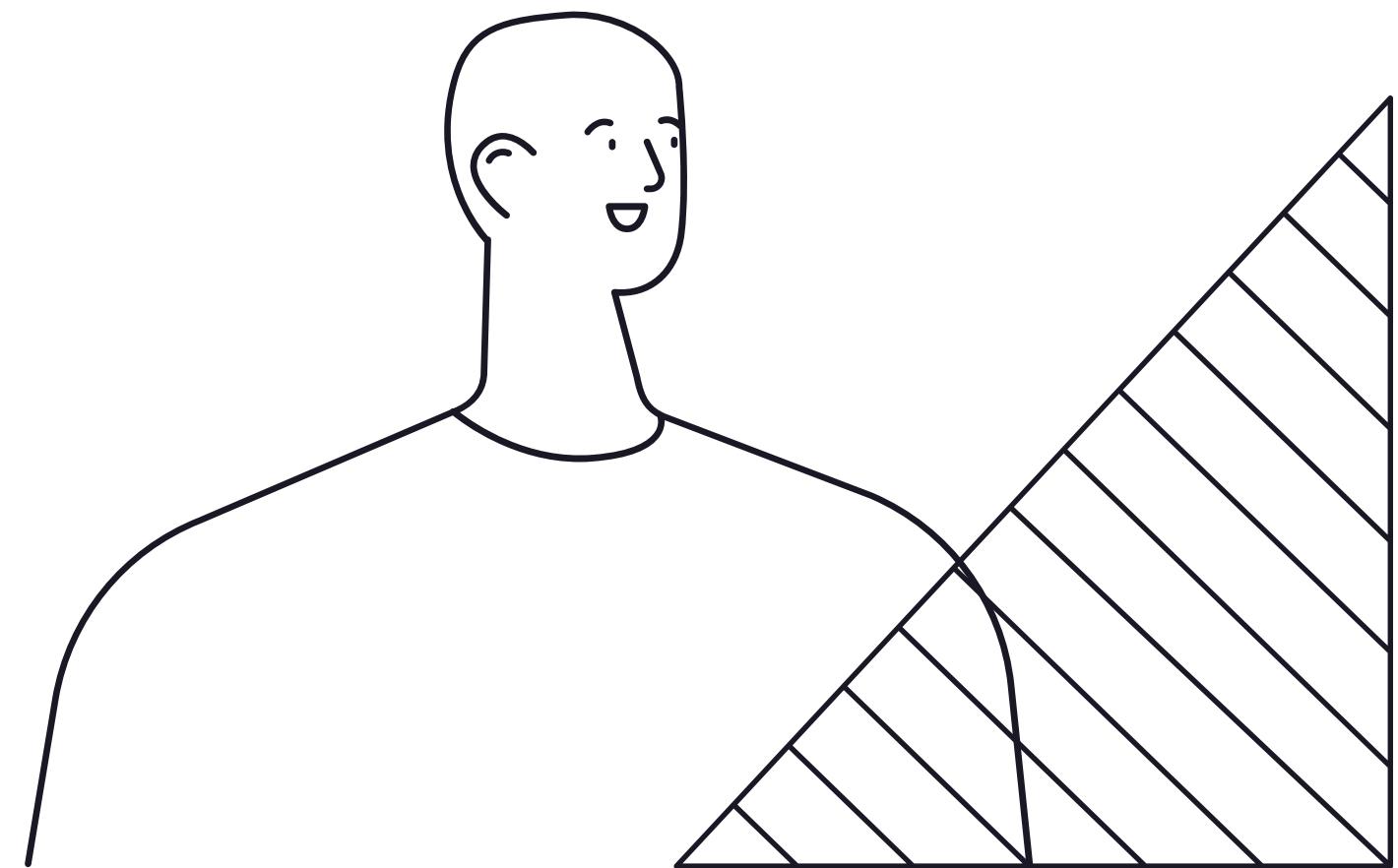


**...with this wisdom and arsenal,
programmers listen to the code, and
they interpret what the code is saying
by abstracting its 'words' into phrases
and concepts which are, in fact,
design principle violations.**

[1]

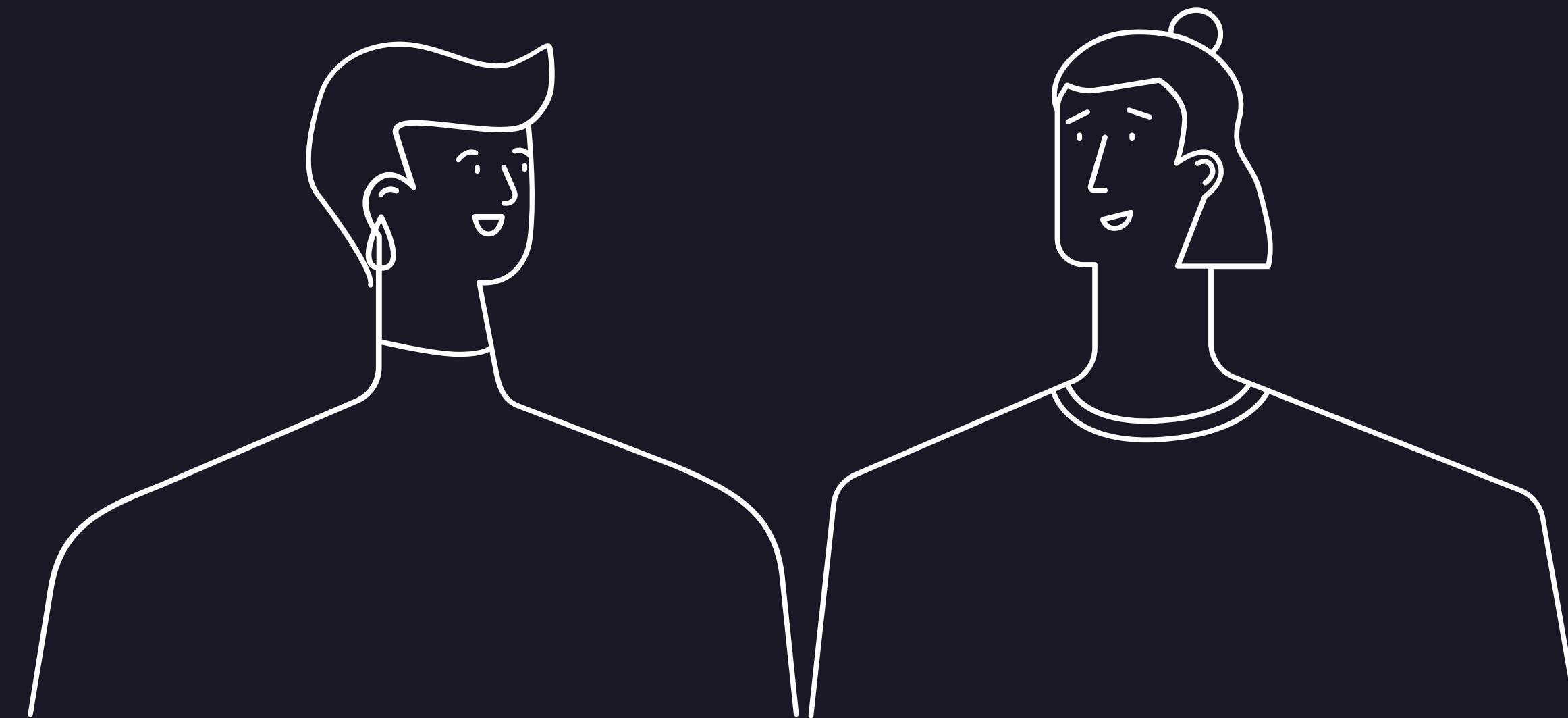
**Solving these changes require a good
understanding of how designs are
implemented. You cannot quench
someones thirst without knowing
what water is.**

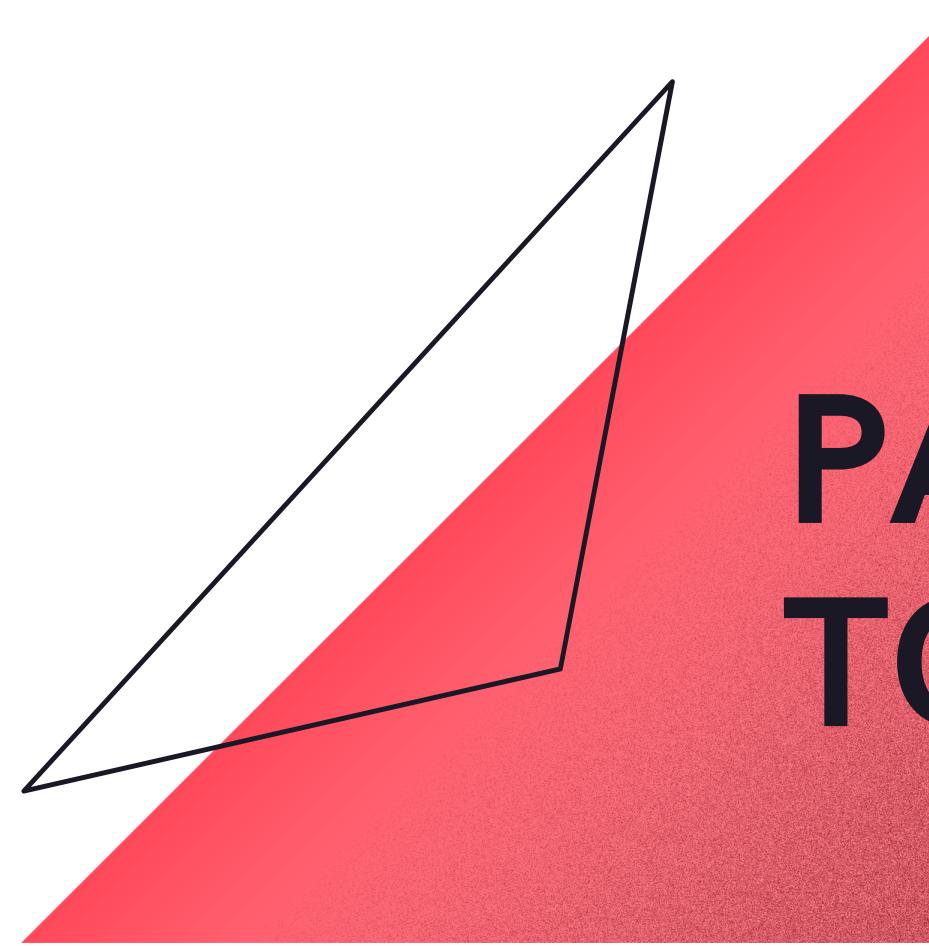
WHAT DOES IT SAY?



WHAT ARE THESE DESIGN PATTERNS

Tested, proven development paradigms, used for speeding up the development process and code readability



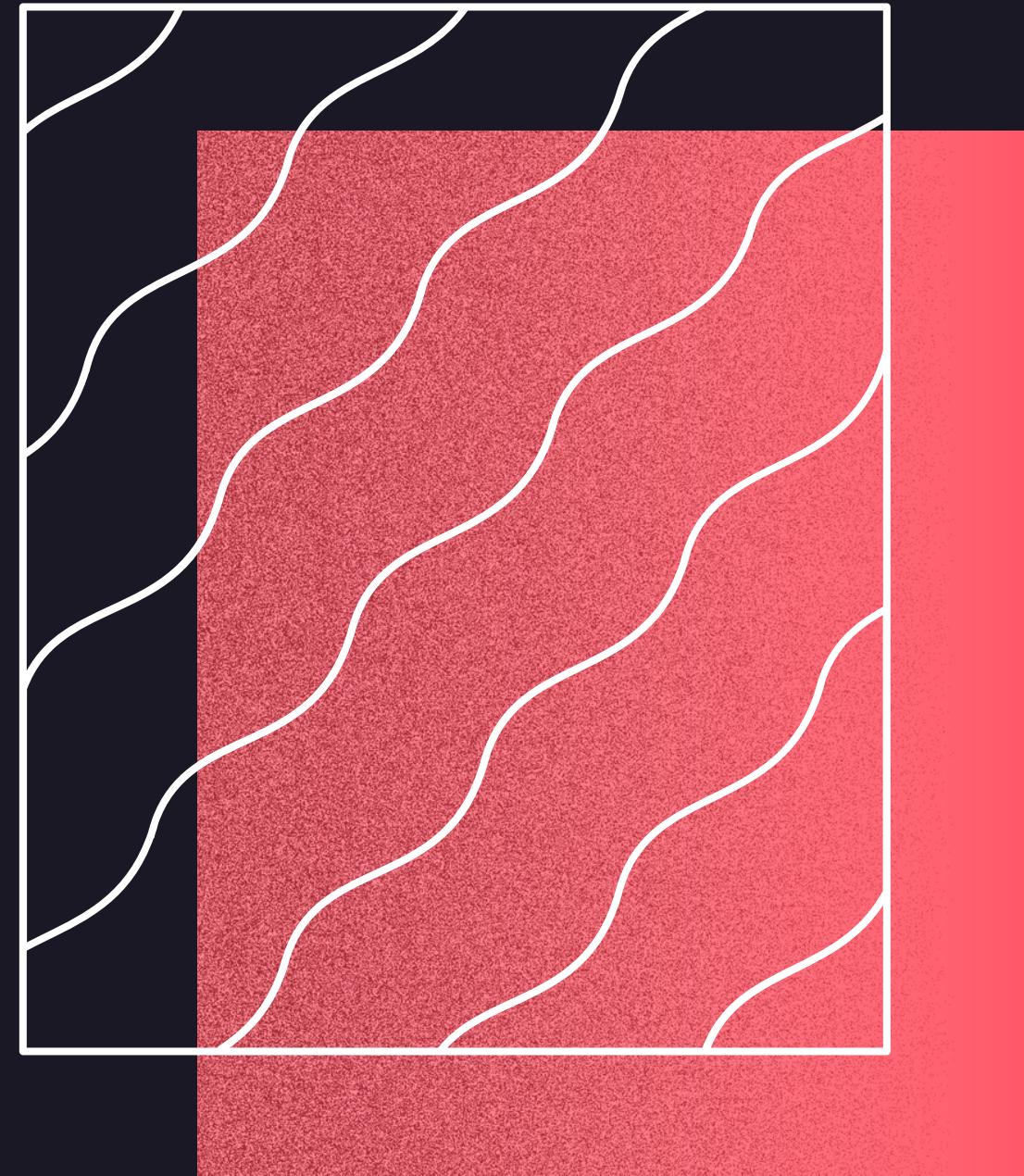


PATTERNS OF TODAY'S LECTURE

**STRATEGY | DECORATOR
PATTERN | PATTERN**

ENCAPSULATING ALGORITHMS FOR CODE REUSE

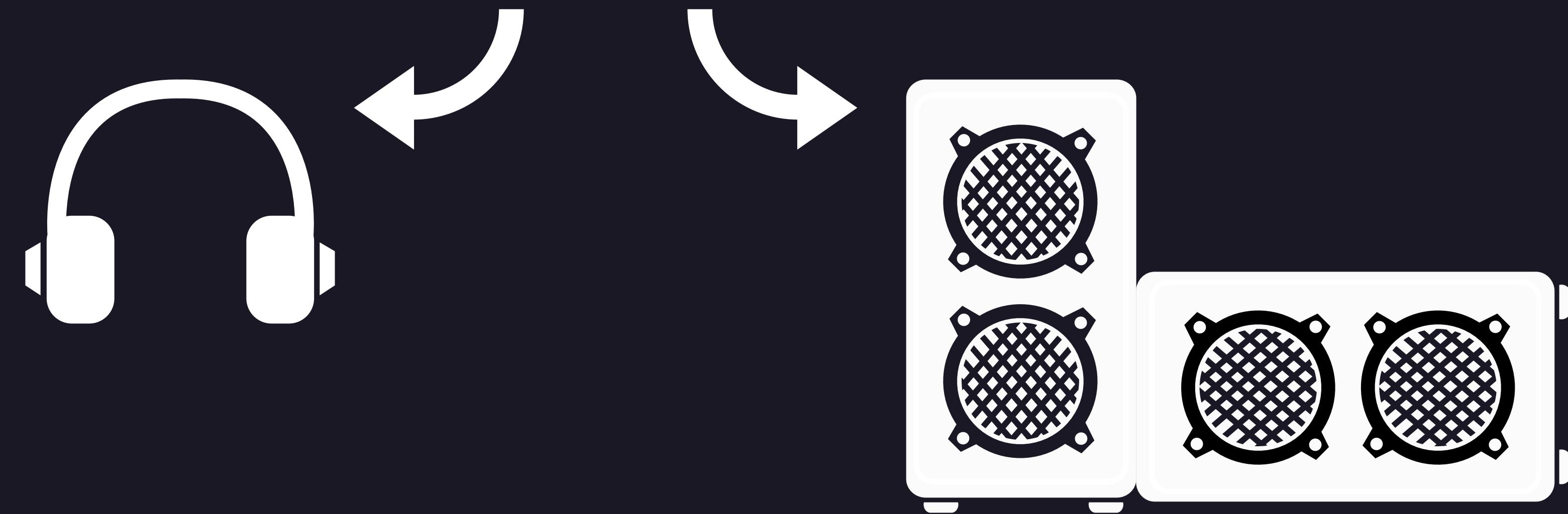
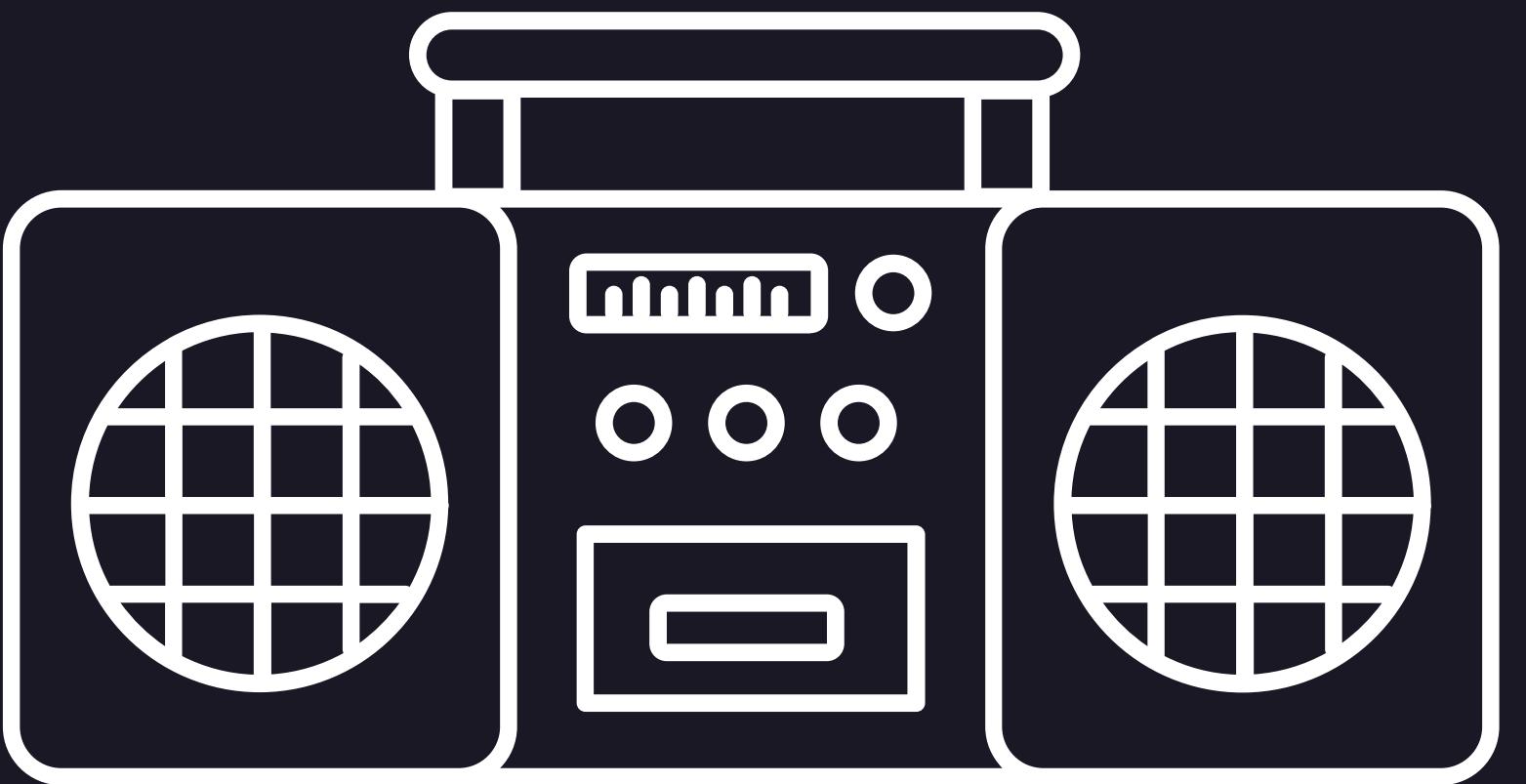
EXTENDING ALGORITHMS FOR CODE REUSE

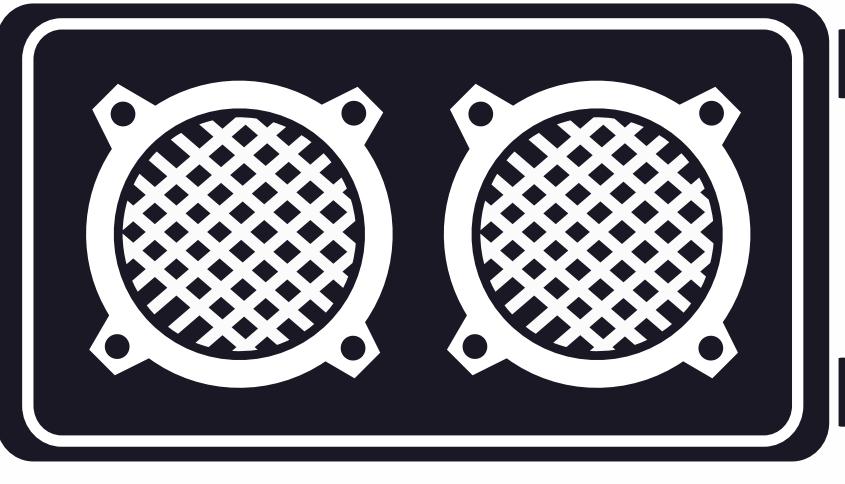
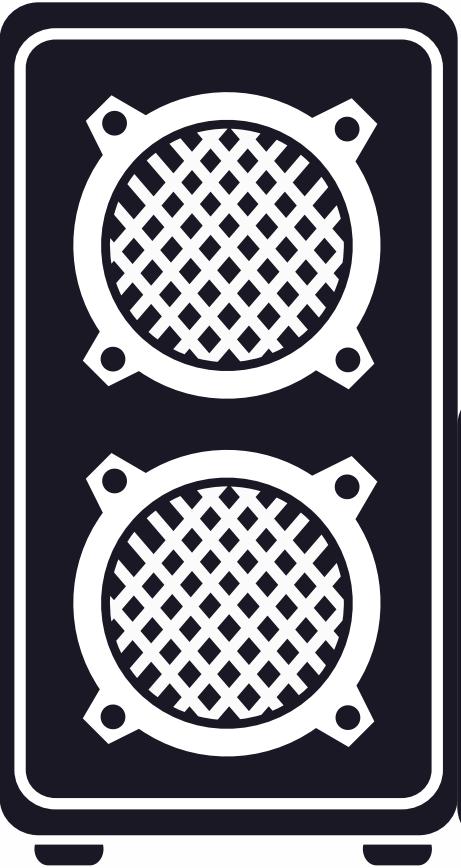
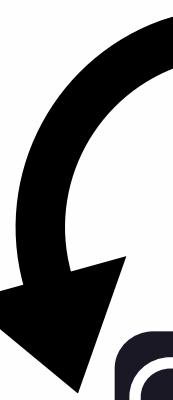
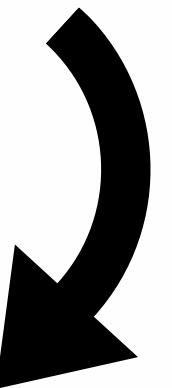
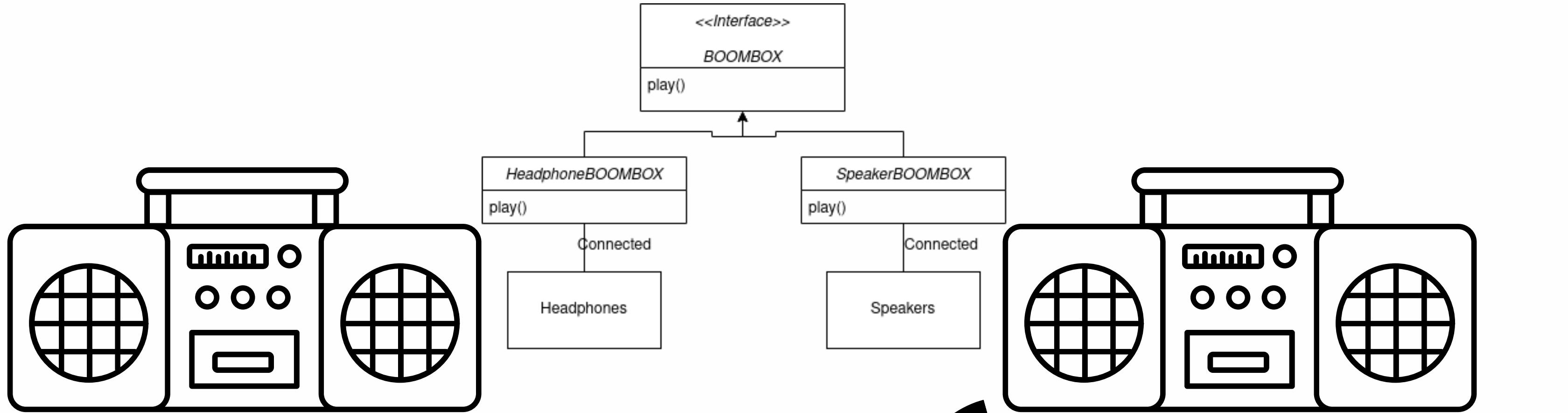


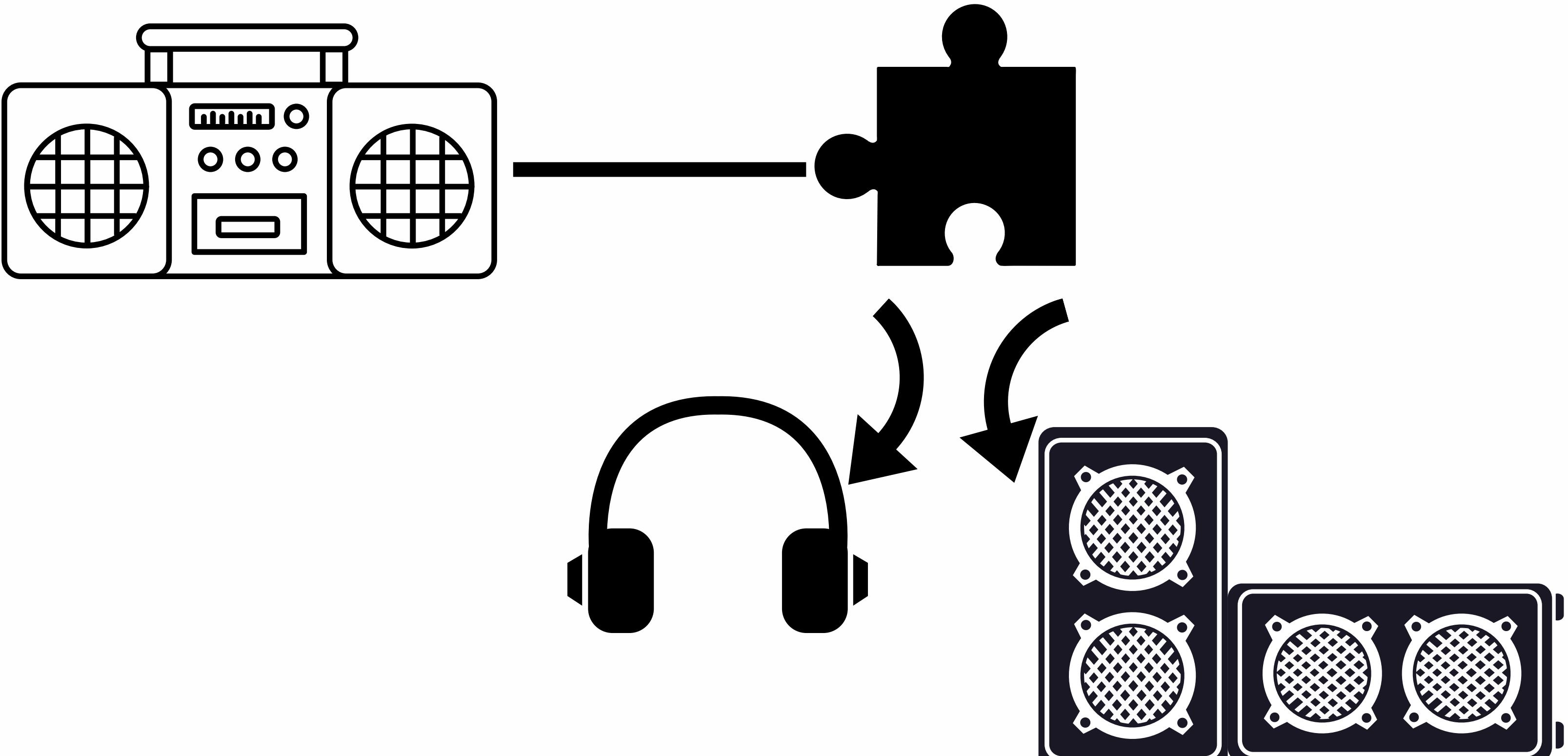
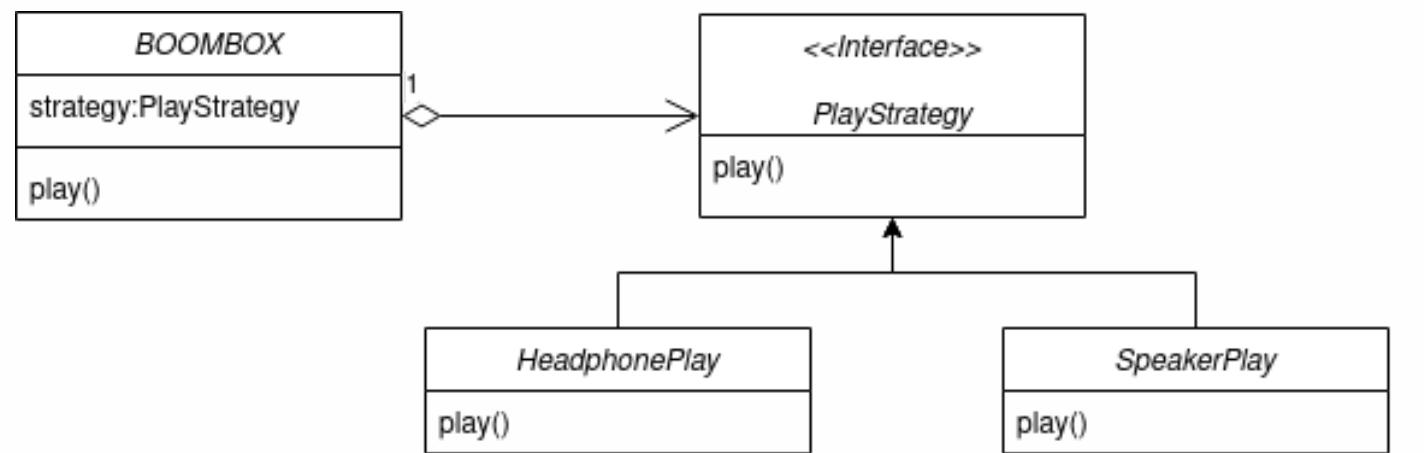
STRATEGY PATTERN

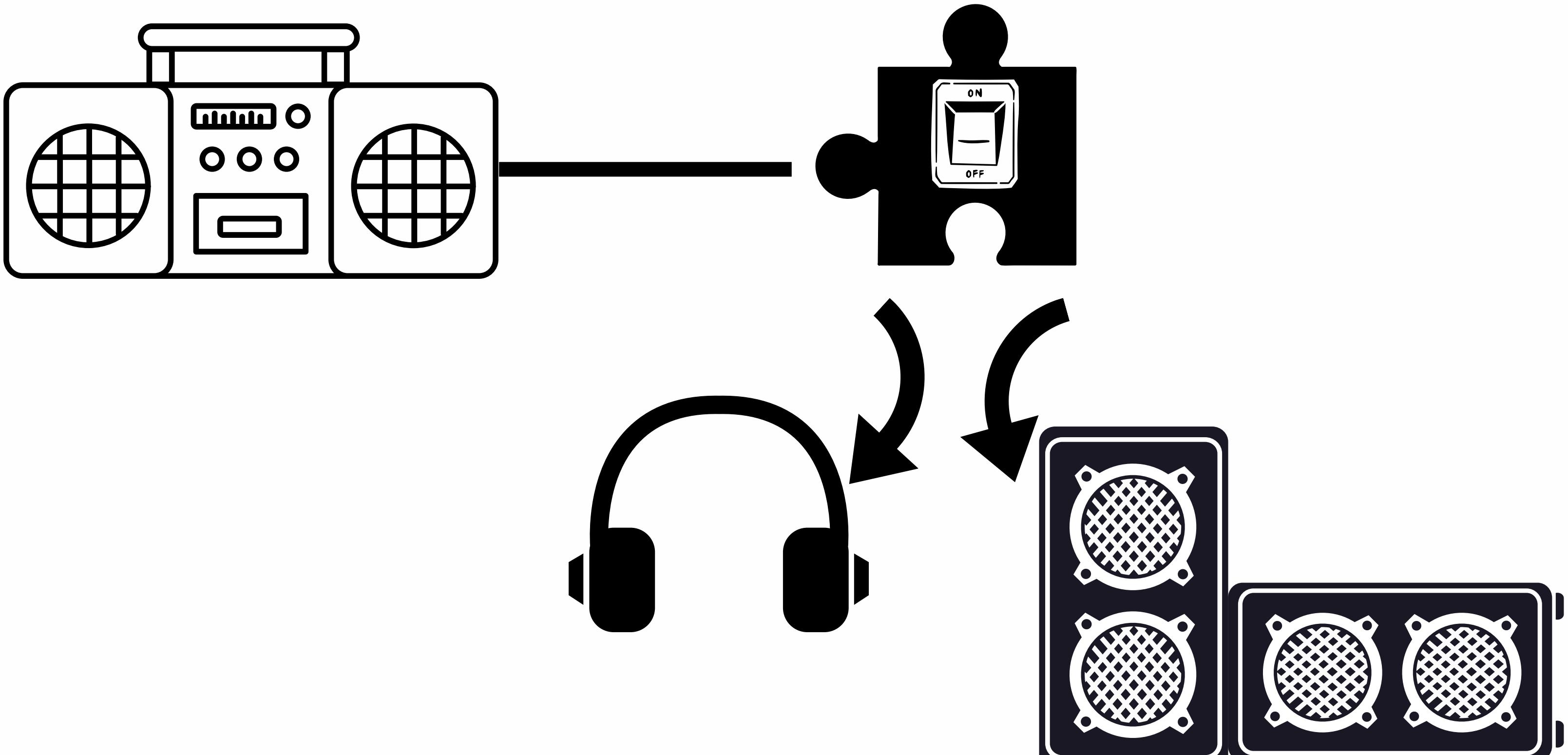
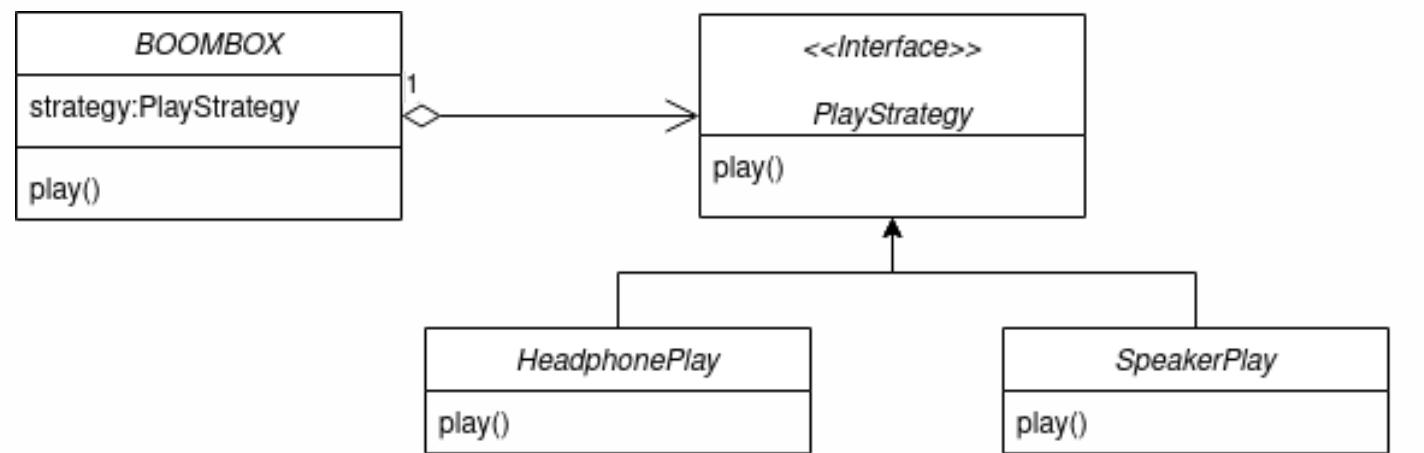
Strategy pattern encapsulates algorithms and so that you can choose algorithm in runtime. It also allows code reuse by allowing multiple agents with similar properties differentiating from implementations to be grouped under the same tent.

A good example can be the music player system.







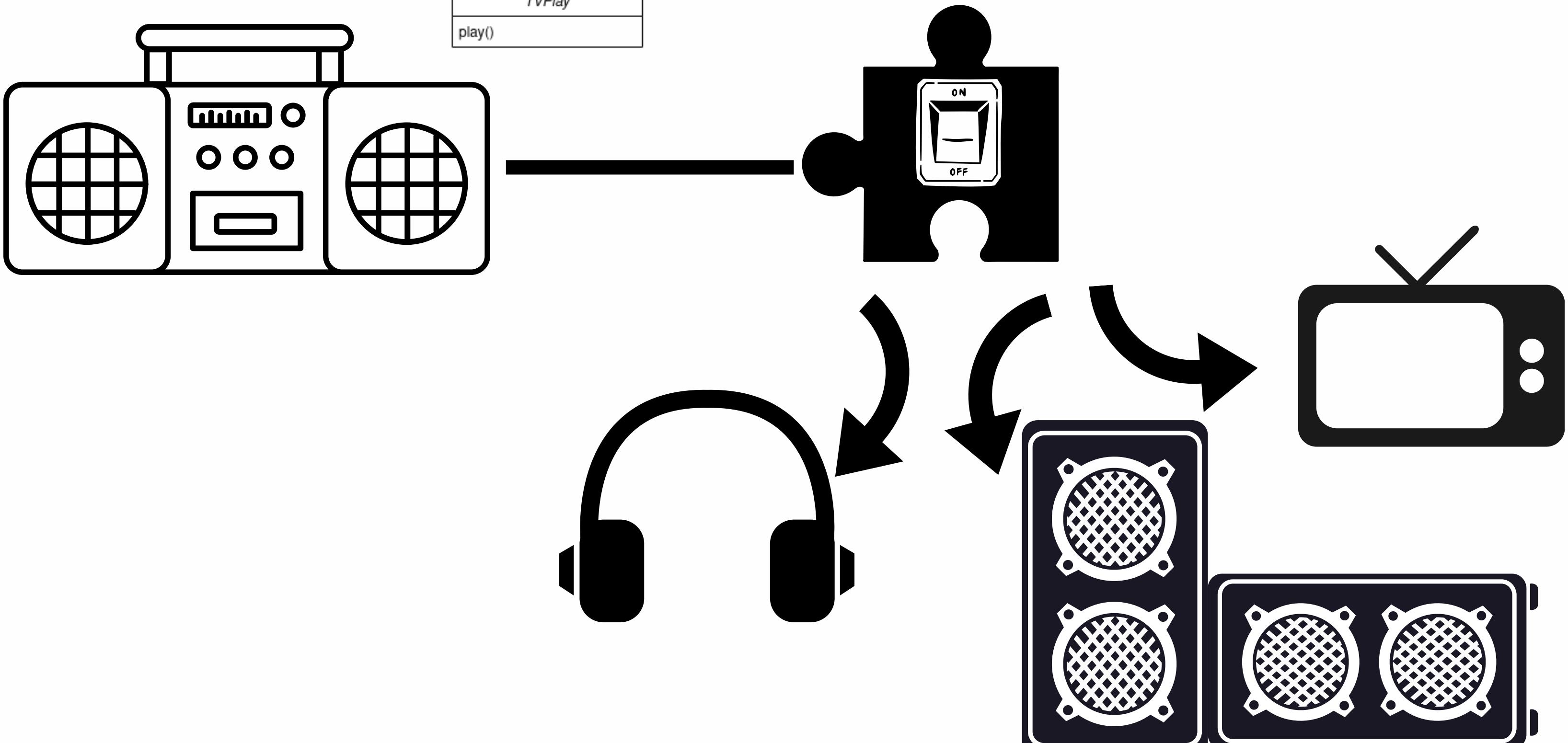
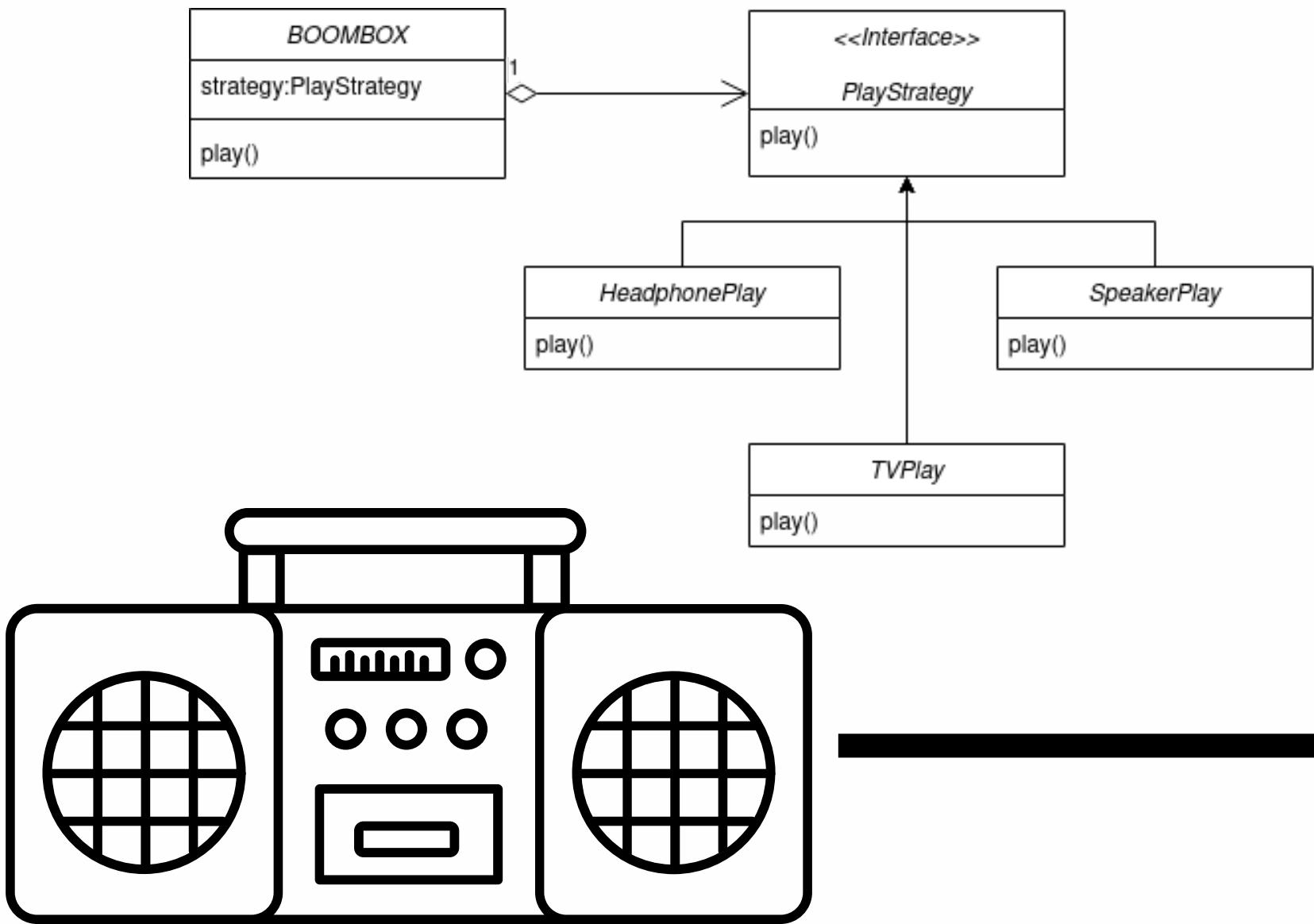


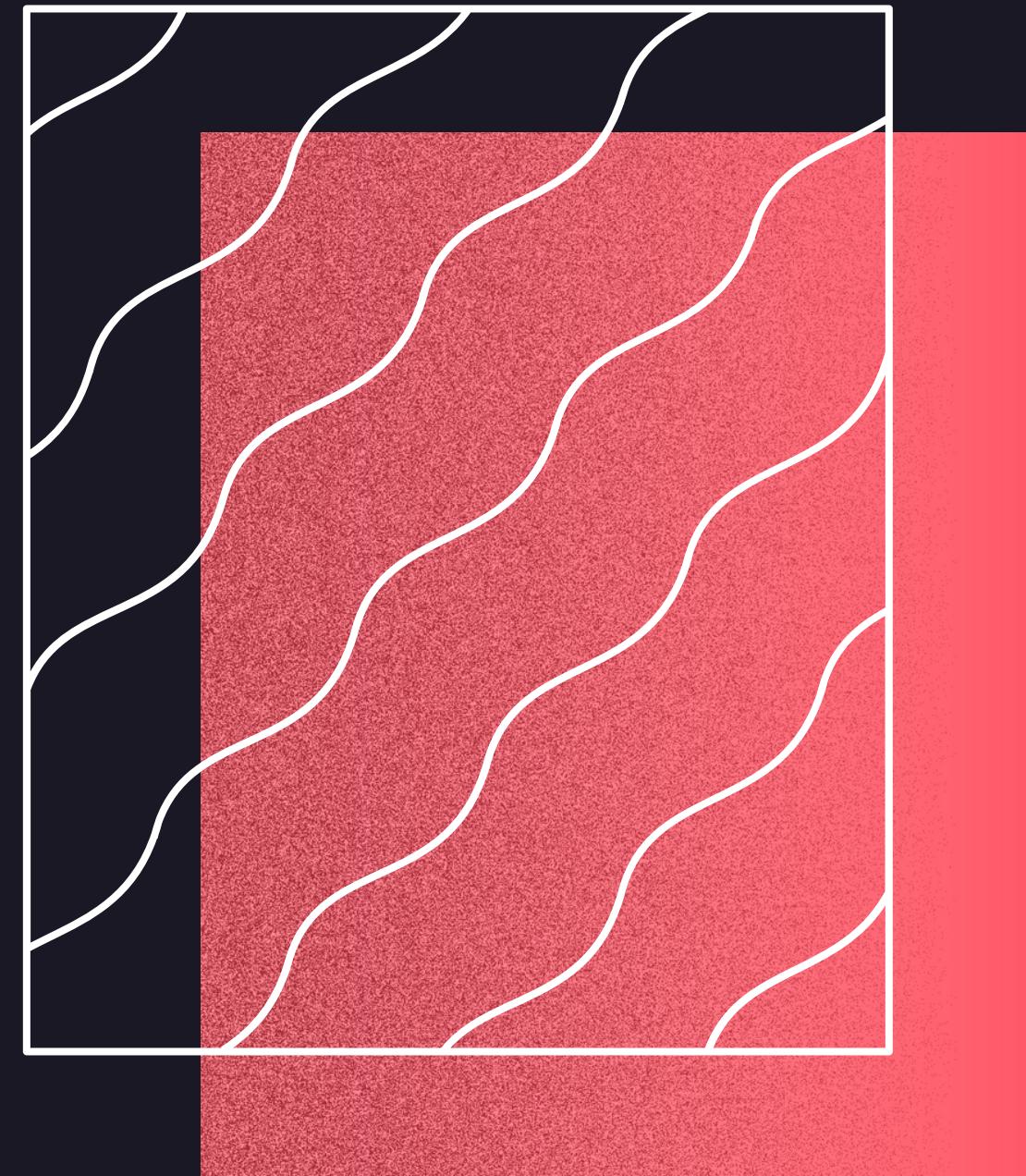
```
1 public interface Boombox {  
2     SoundSignal music;  
3     void play();  
4 }  
5  
6 public class HeadphoneBoombox extends Boombox{  
7     void play(){  
8         //I am playing this on headphones  
9         Headphones.play(this.music)  
10    }  
11 }
```

without design patterns

```
13 public interface PlayStrategy{  
14     void play(SoundSignal);  
15 }  
16  
17 public class Boombox{  
18     SoundSignal music;  
19     PlayStrategy playStrategy;  
20     public Boombox(PlayStrategy playStrategy){  
21         this.playStrategy = playStrategy;  
22     }  
23     void play(){  
24         playStrategy(music);  
25     }  
26 }  
27  
28 public class HeadphonePlayStrategy implements PlayStrategy{  
29     //you can get the required stuff as parameters  
30     void play(SoundSignal music){  
31         //play on headphones  
32         Headphone.play(music);  
33     }  
34 }
```

with design patterns

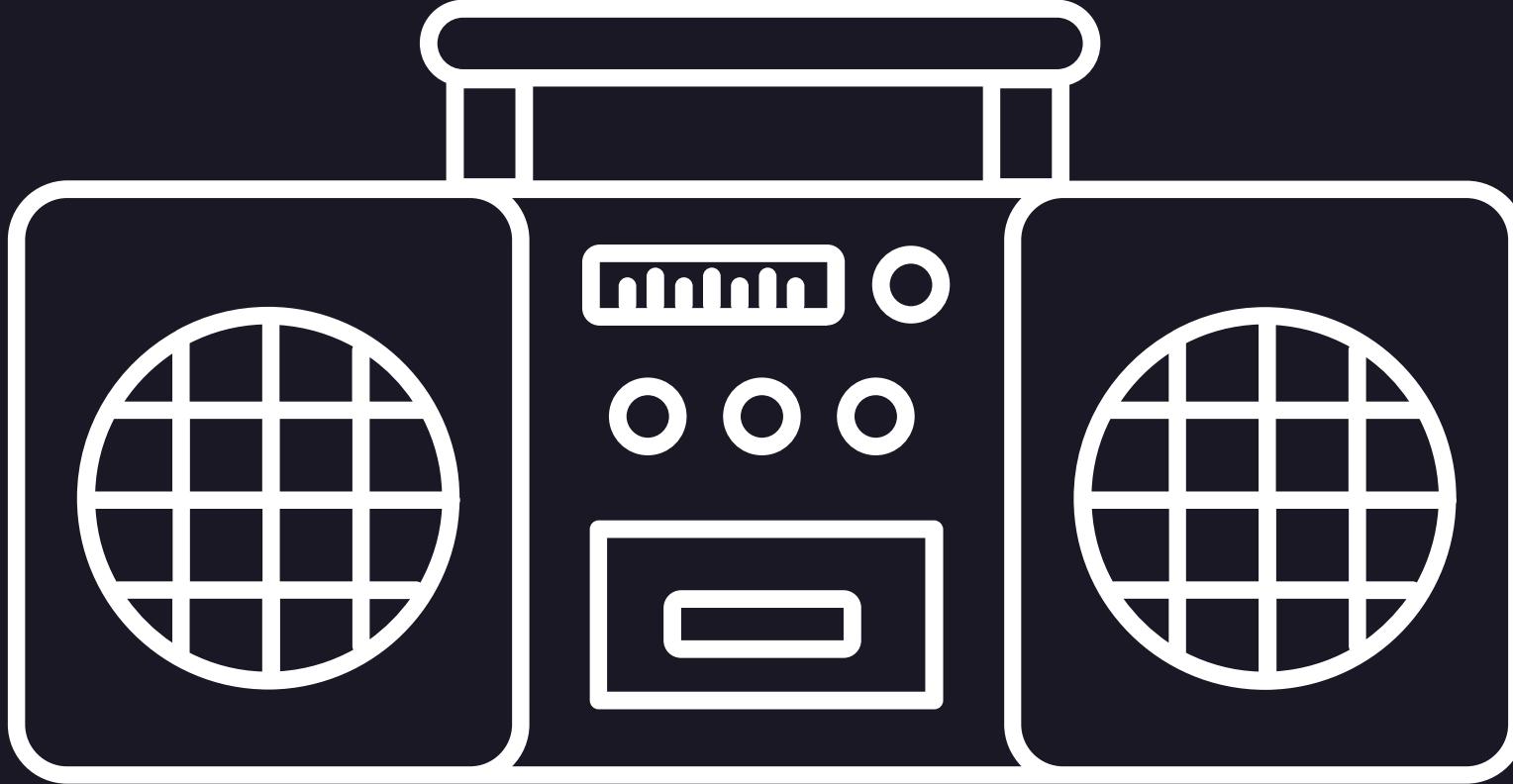




DECORATOR PATTERN

Decorator pattern allows you to extend algorithms in order to improve code reuse.

A good example can be again our music player system.



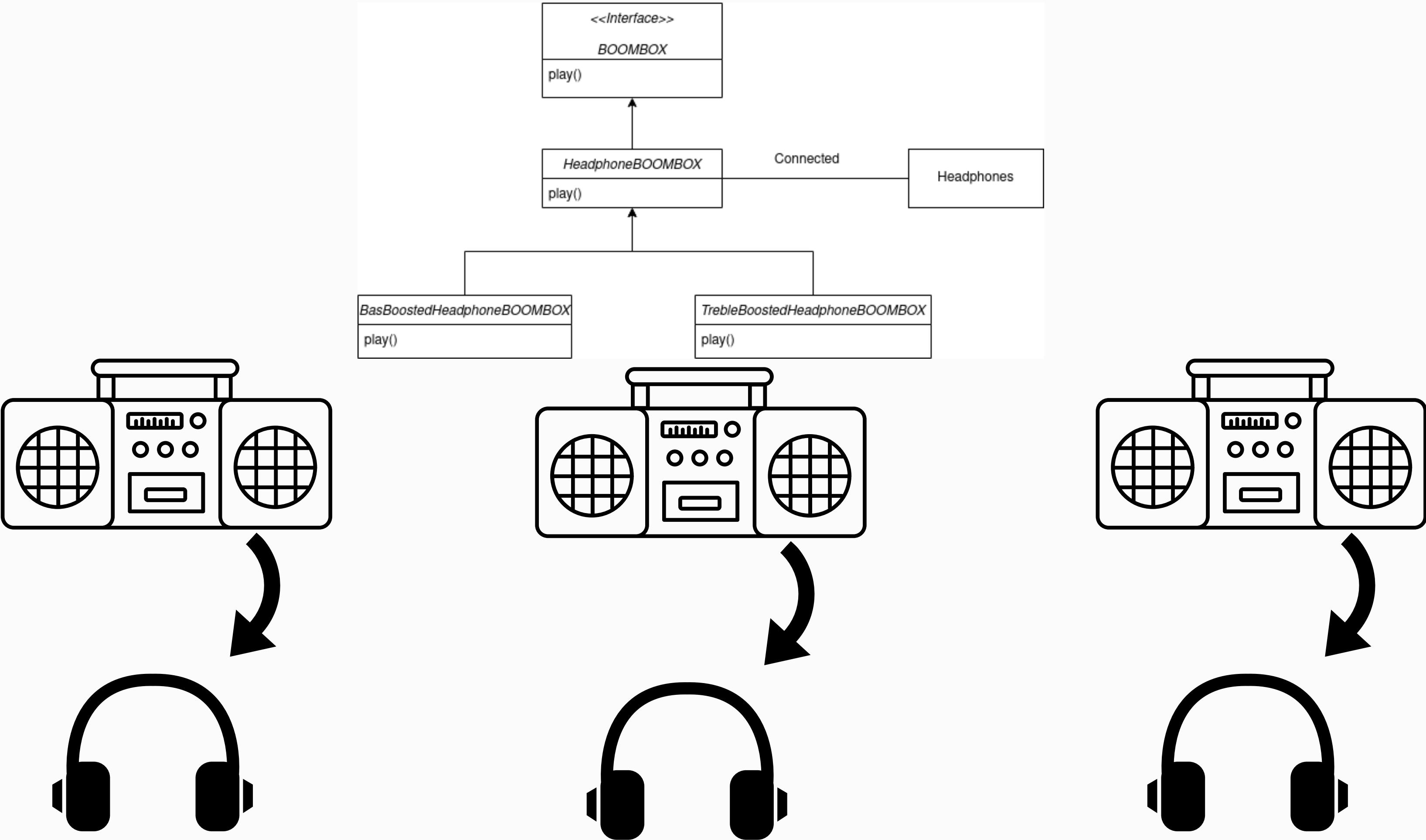
BASS BOOSTED

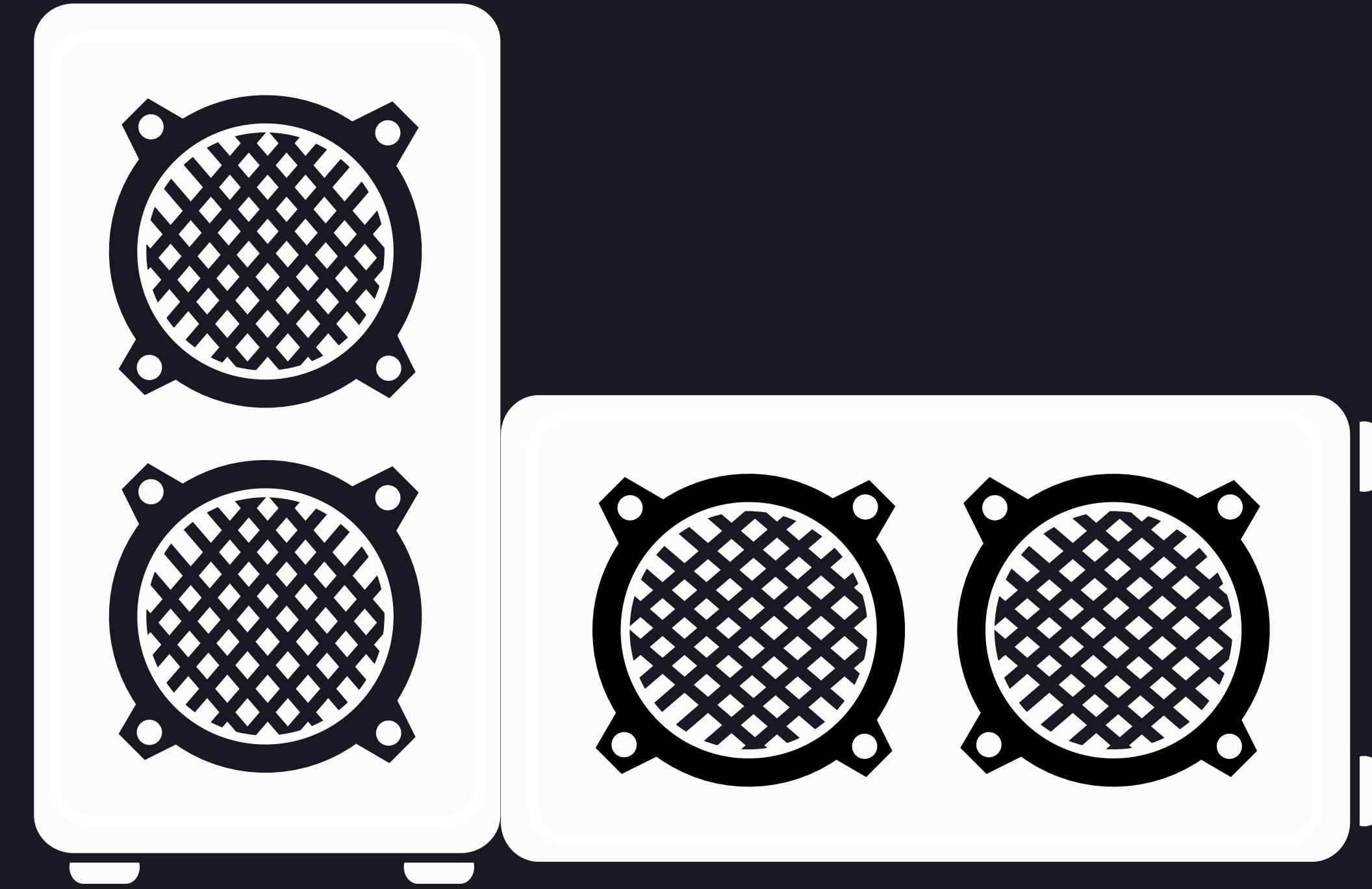


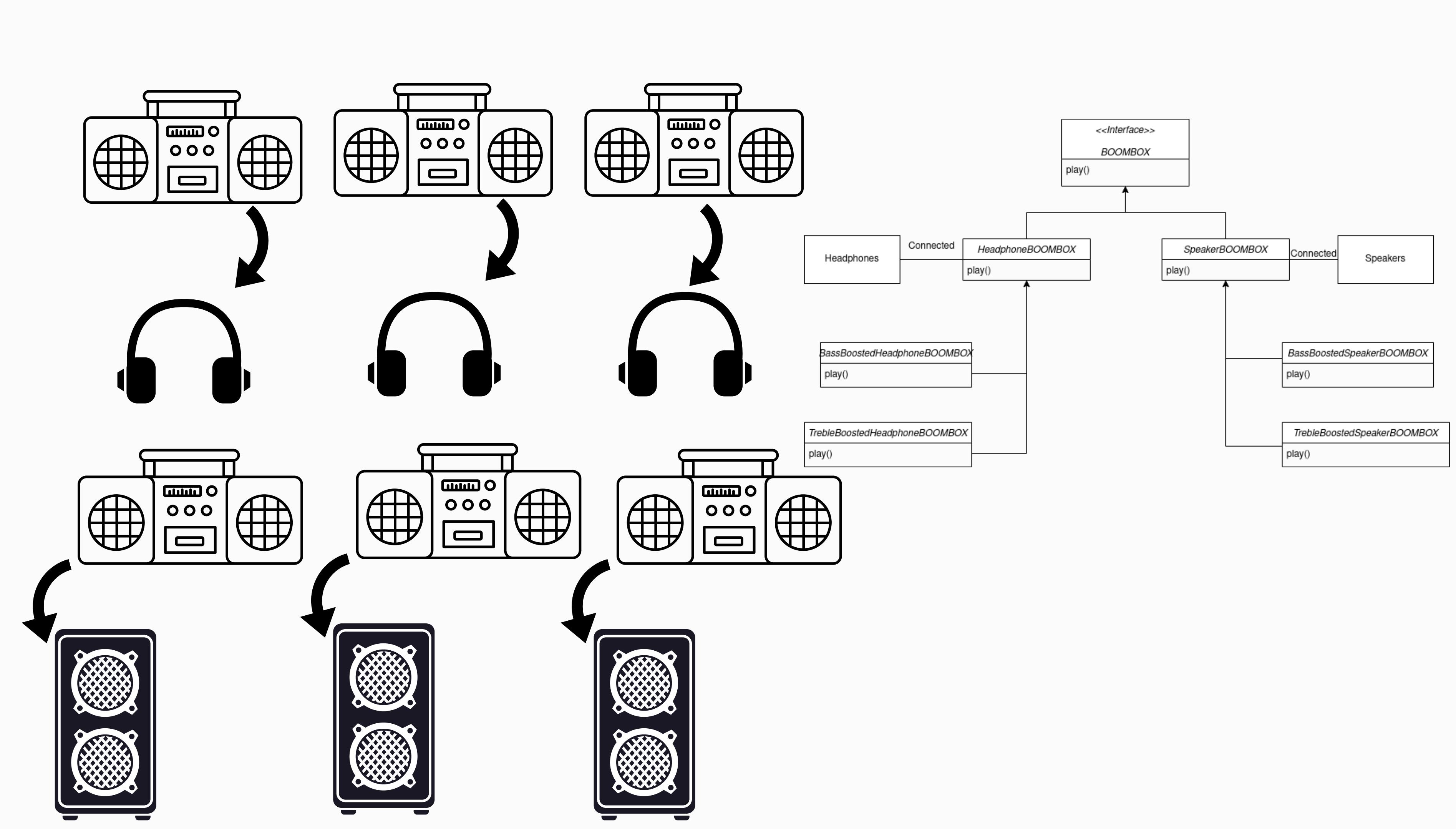
NORMAL

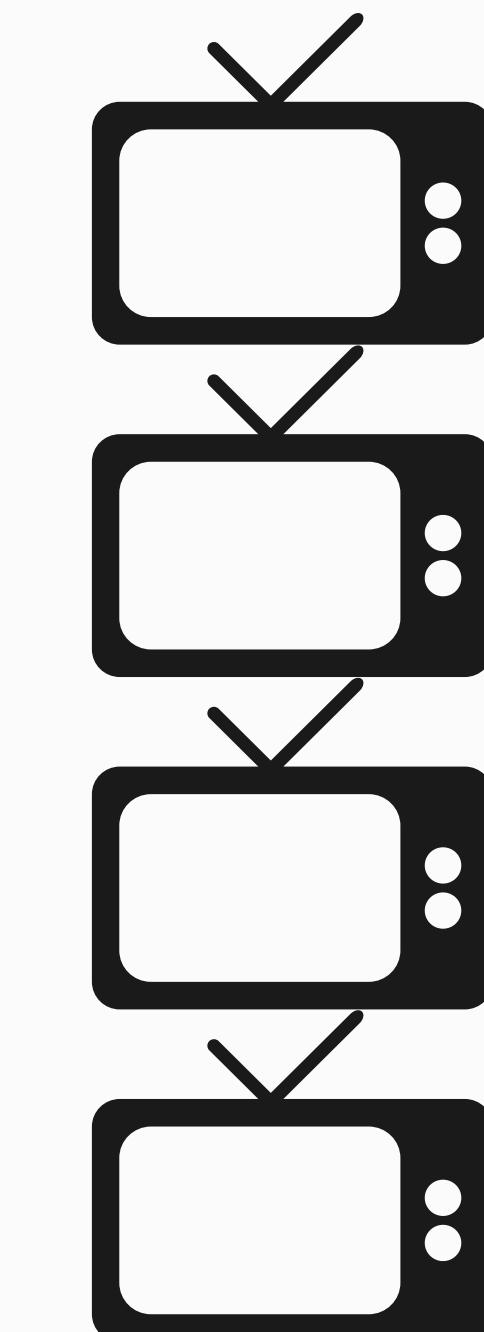
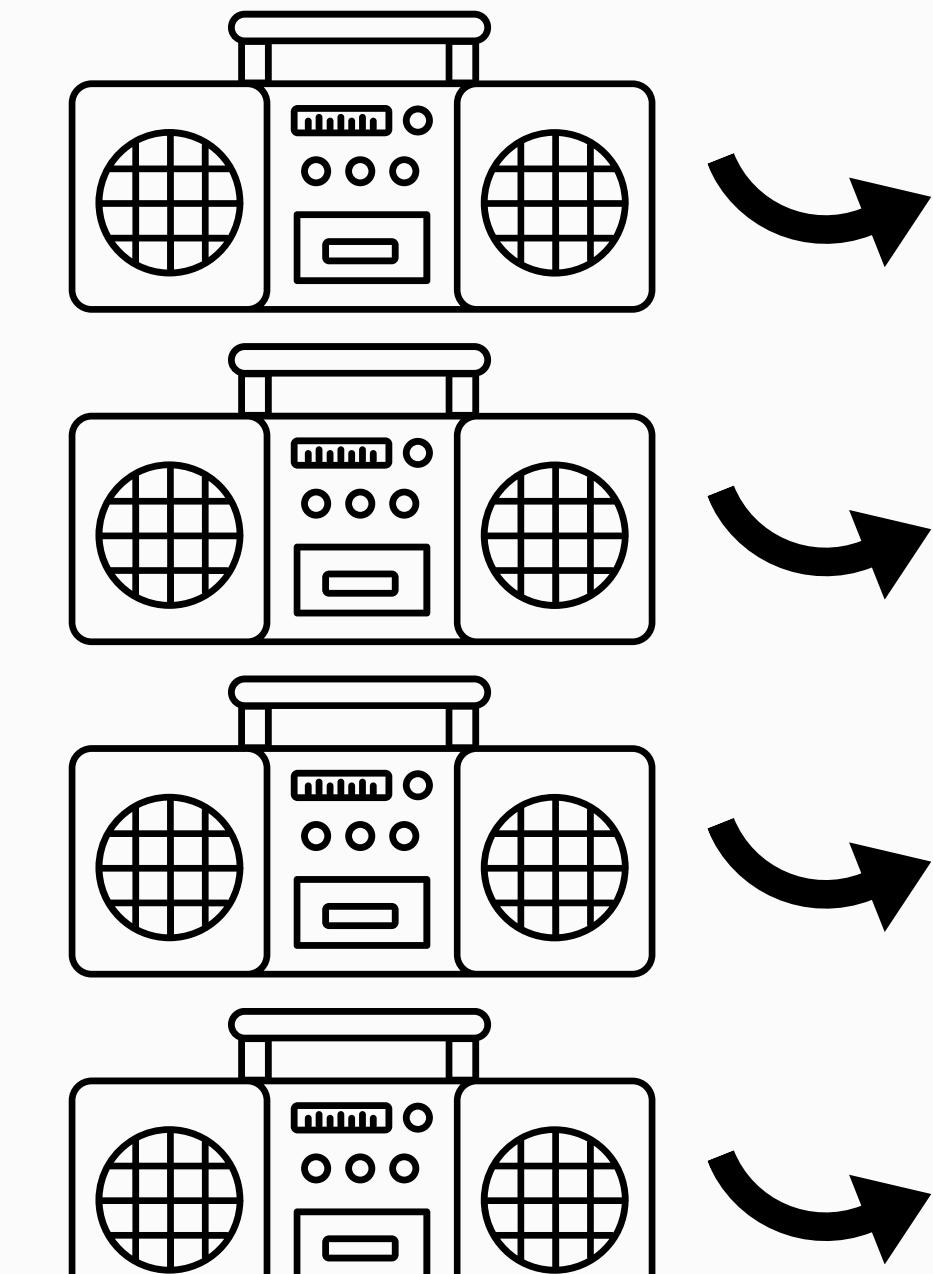
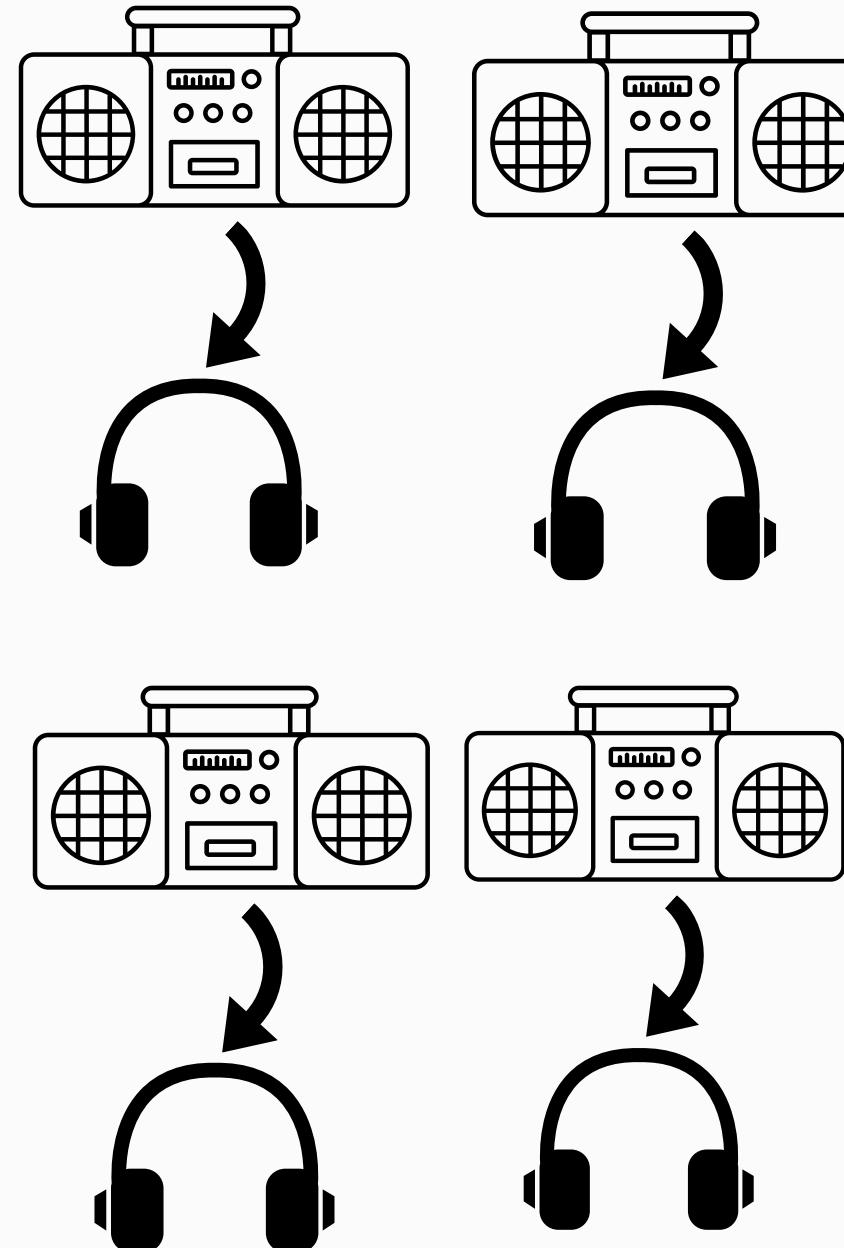
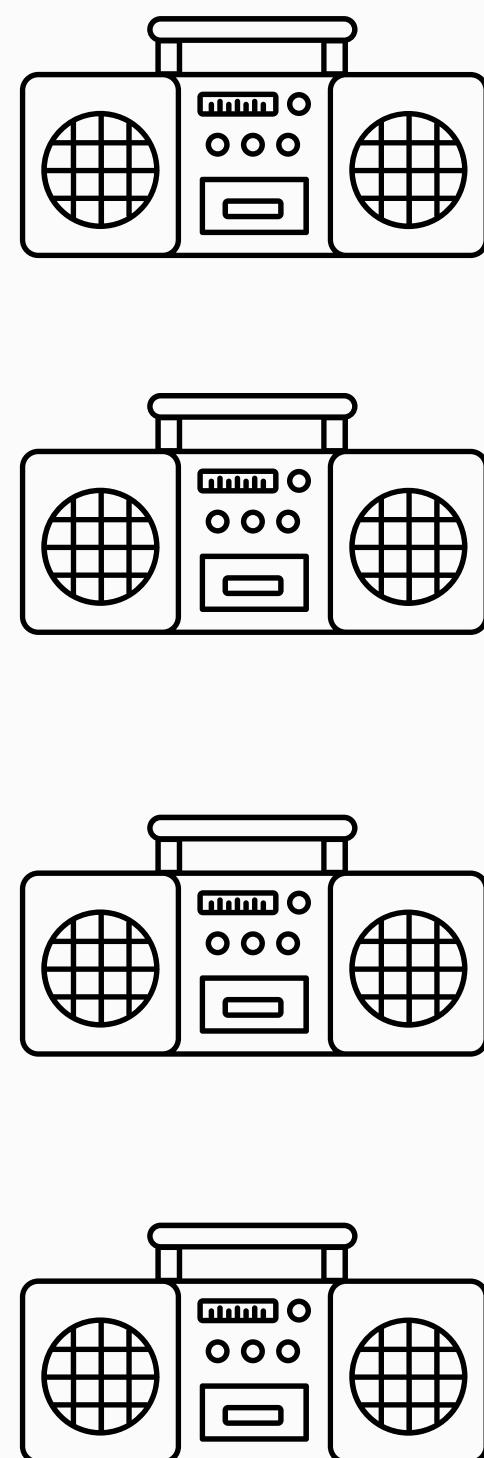


TREBLE BOOSTED







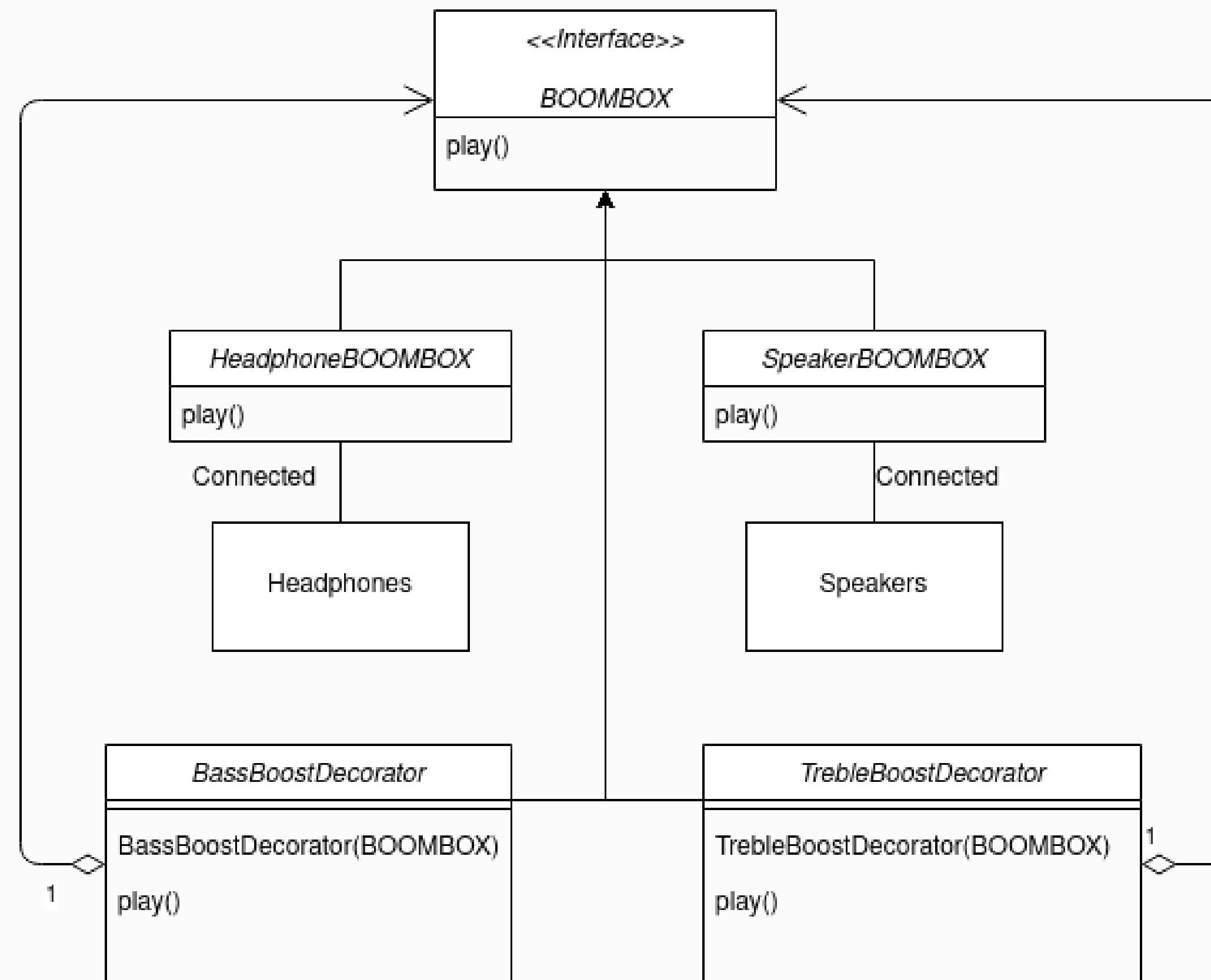


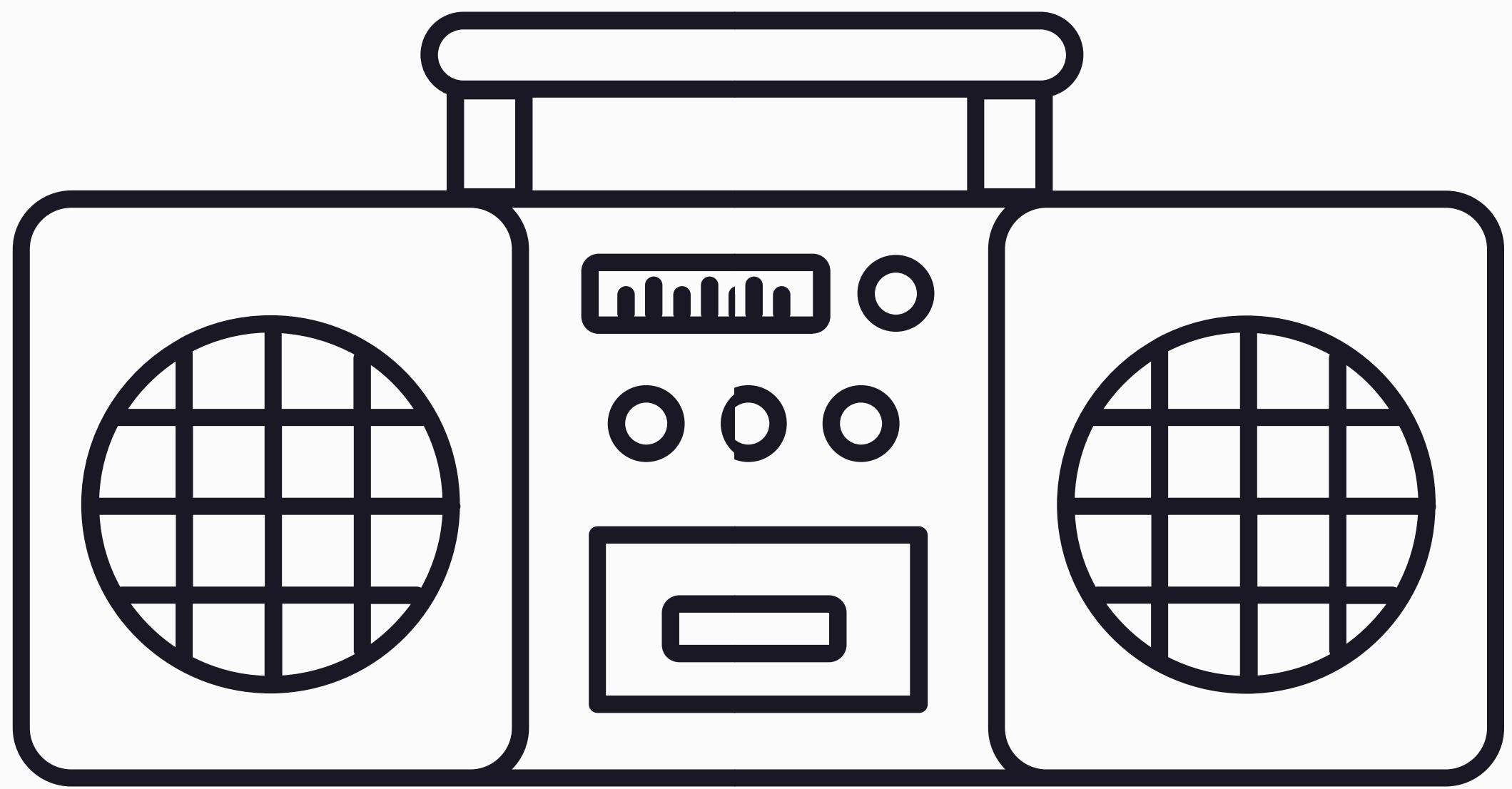


SO WHAT SHOULD WE DO?



Decorator pattern allows us to encapsulate these type of behaviors so that it is easier to extend algorithms.





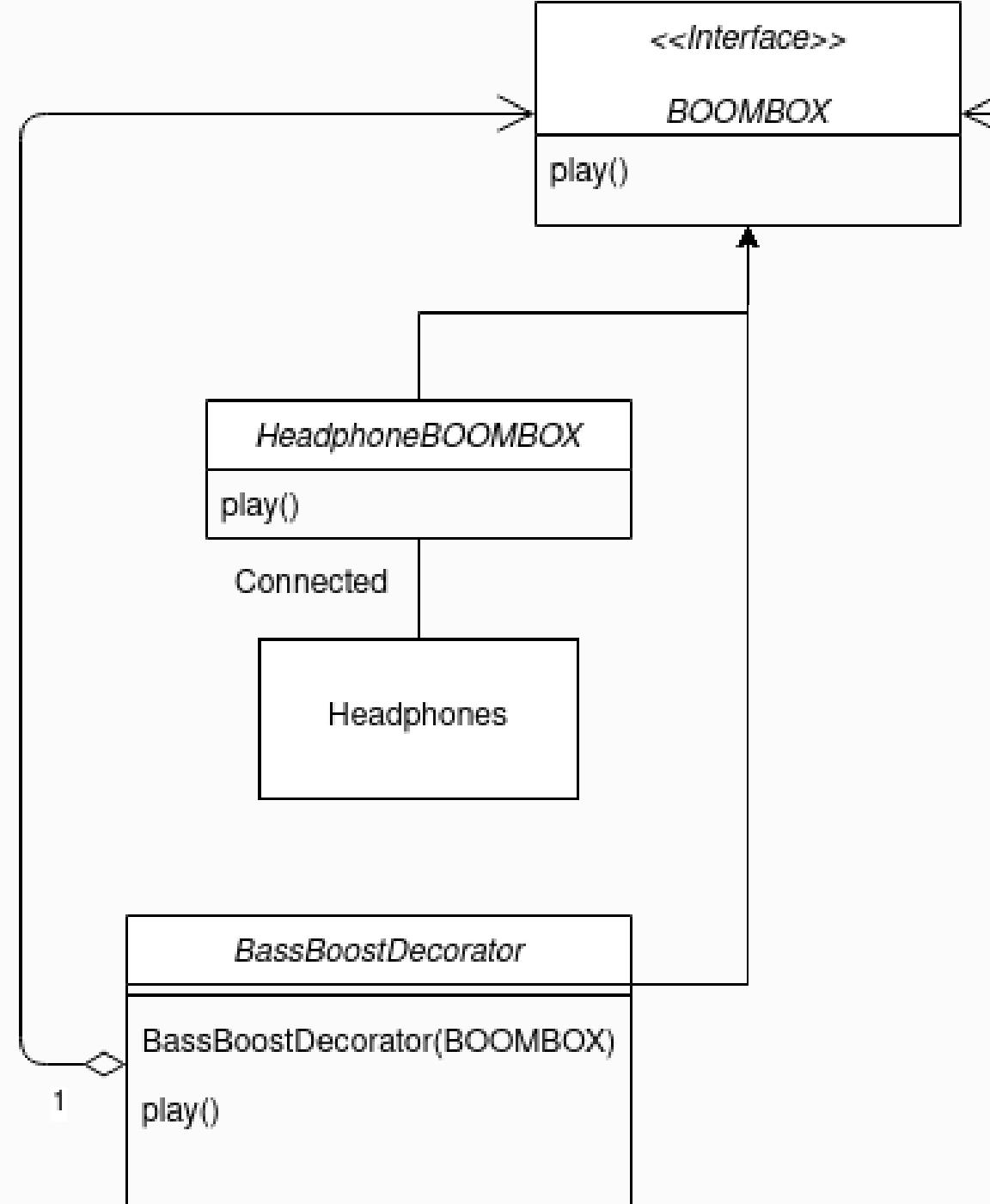
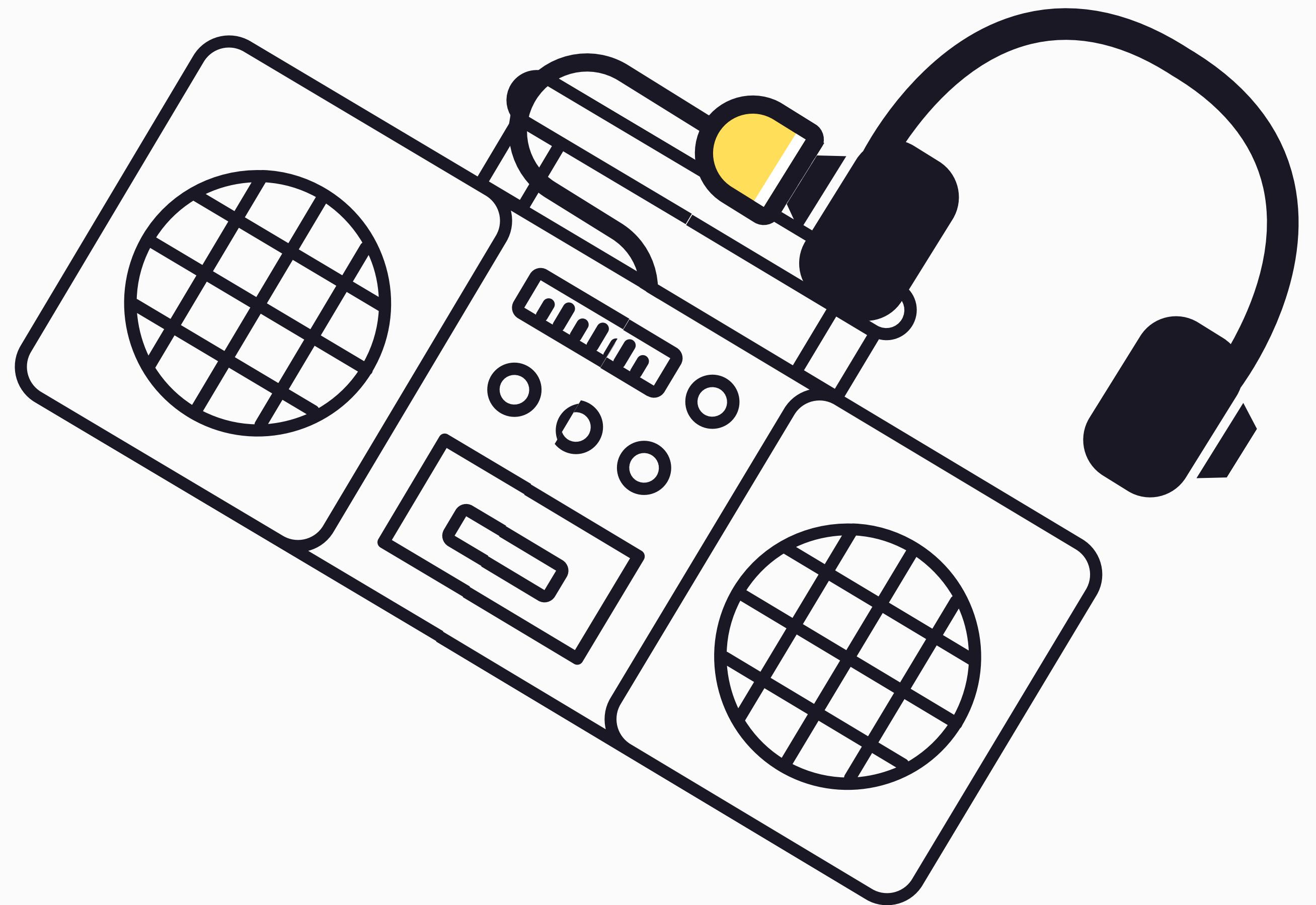


HeadphoneBOOMBOX

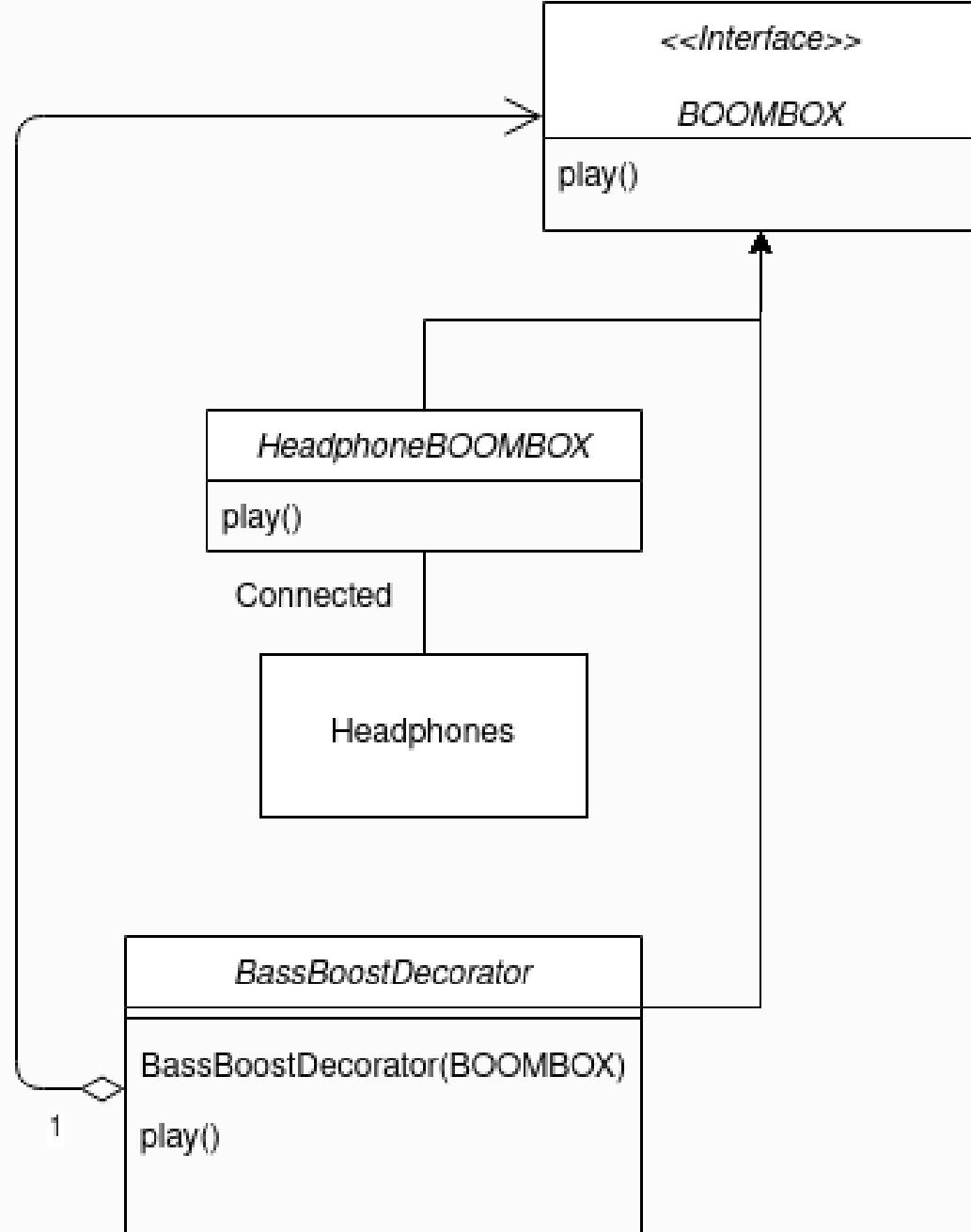
play()

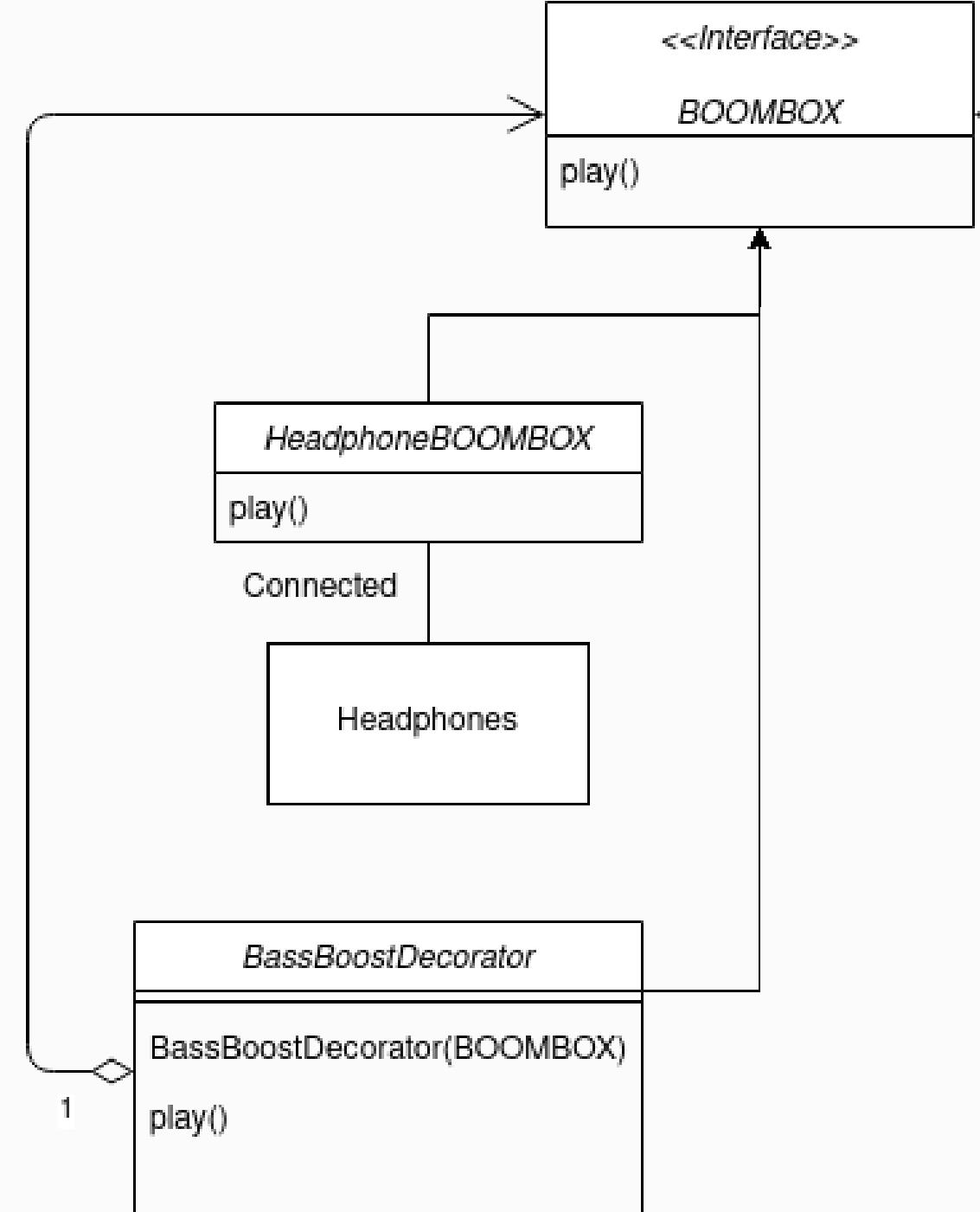
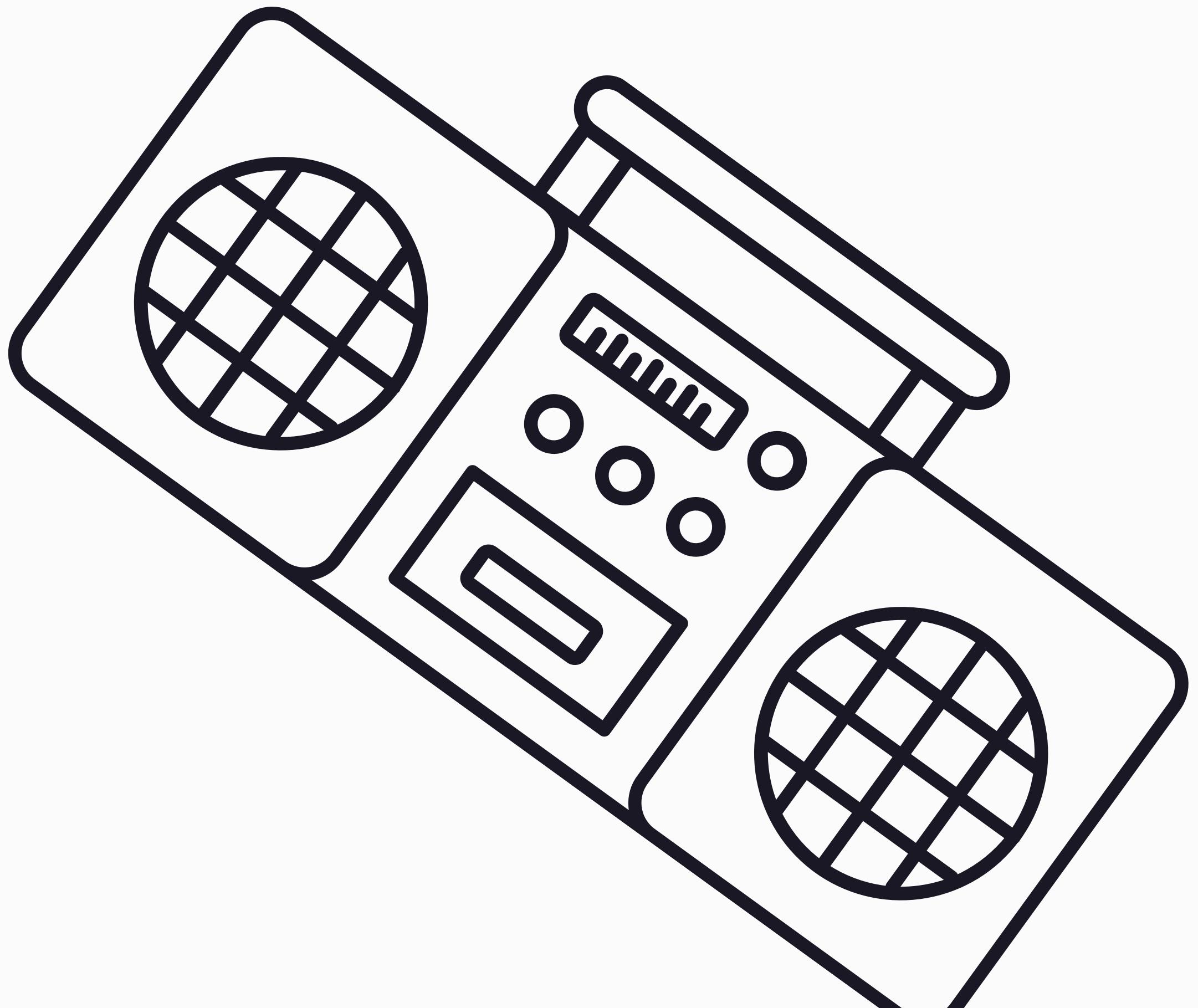
Connected

Headphones









```
1 public interface Boombox {  
2     SoundSignal music;  
3     void play();  
4 }  
5  
6 public class HeadphoneBoombox implements| Boombox{  
7     void play(){  
8         //I am playing this on headphones  
9         Headphones.play(this.music)  
10    }  
11 }  
12  
13 public class BassBoostedHeadphoneBoombox extends HeadphoneBoombox{  
14     void play(){  
15         this.music.bases++;  
16         super.play();  
17     }  
18 }
```

```
1 public interface Boombox {  
2     SoundSignal music;  
3     void play();  
4 }  
5  
6 public class HeadphoneBoombox implements Boombox{  
7     void play(){  
8         //I am playing this on headphones  
9         Headphones.play(this.music)  
10    }  
11 }  
12  
13 public class BassBoostDecorator extends Boombox{  
14     Boombox boombox;  
15     public BassBoostDecorator(Boombox boombox){  
16         this.boombox = boombox;  
17     }  
18     void play(){  
19         boombox.music.bases++;  
20         boombox.play();  
21     }  
22 }
```

Implementation with design pattern

```
1 public interface Boombox {  
2     SoundSignal music;  
3     void play();  
4 }  
5  
6 public class HeadphoneBoombox implements Boombox{  
7     void play(){  
8         //I am playing this on headphones  
9         Headphones.play(this.music)  
10    }  
11 }  
--
```

```
public class BassBoostedHeadphoneBoombox extends HeadphoneBoombox{  
    void play(){  
        this.music.bases++;  
        super.play();  
    }  
}
```

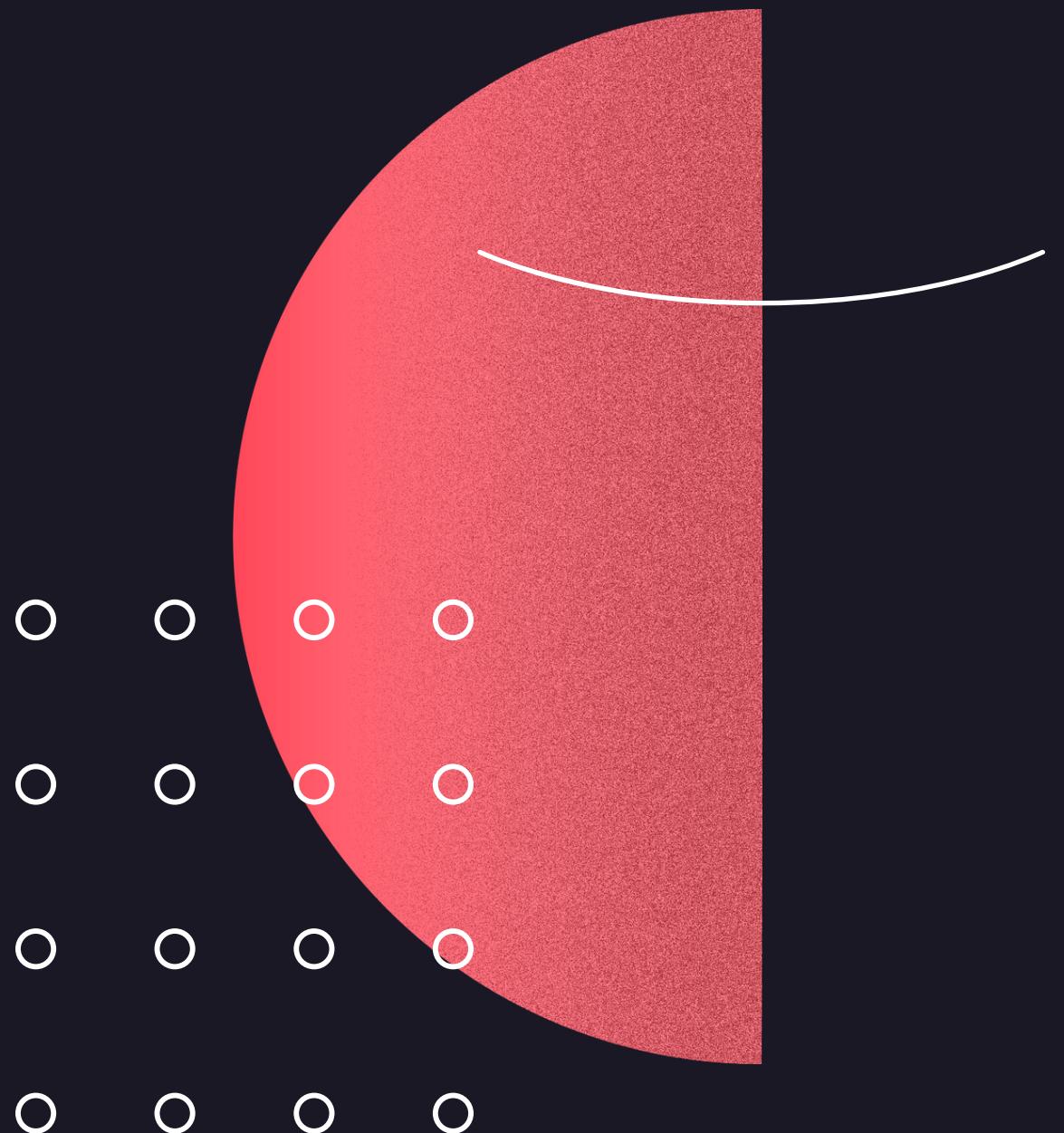
Implementation without design pattern

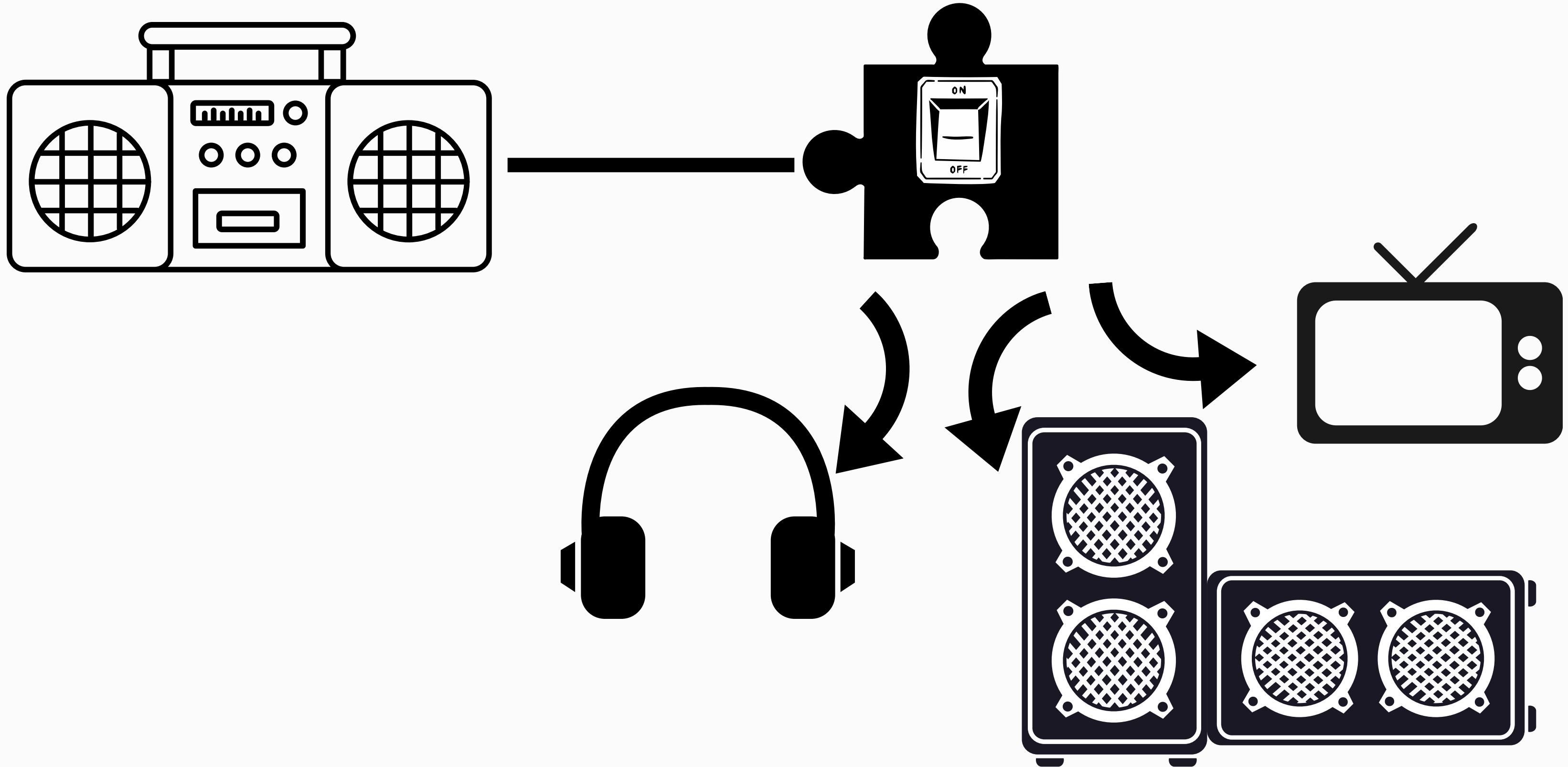
```
12  
13 public class BassBoostDecorator extends Boombox{  
14     Boombox boombox;  
15     public BassBoostDecorator(Boombox boombox){  
16         this.boombox = boombox;  
17     }  
18     void play(){  
19         boombox.music.bases++;  
20         boombox.play();  
21     }  
22 }
```

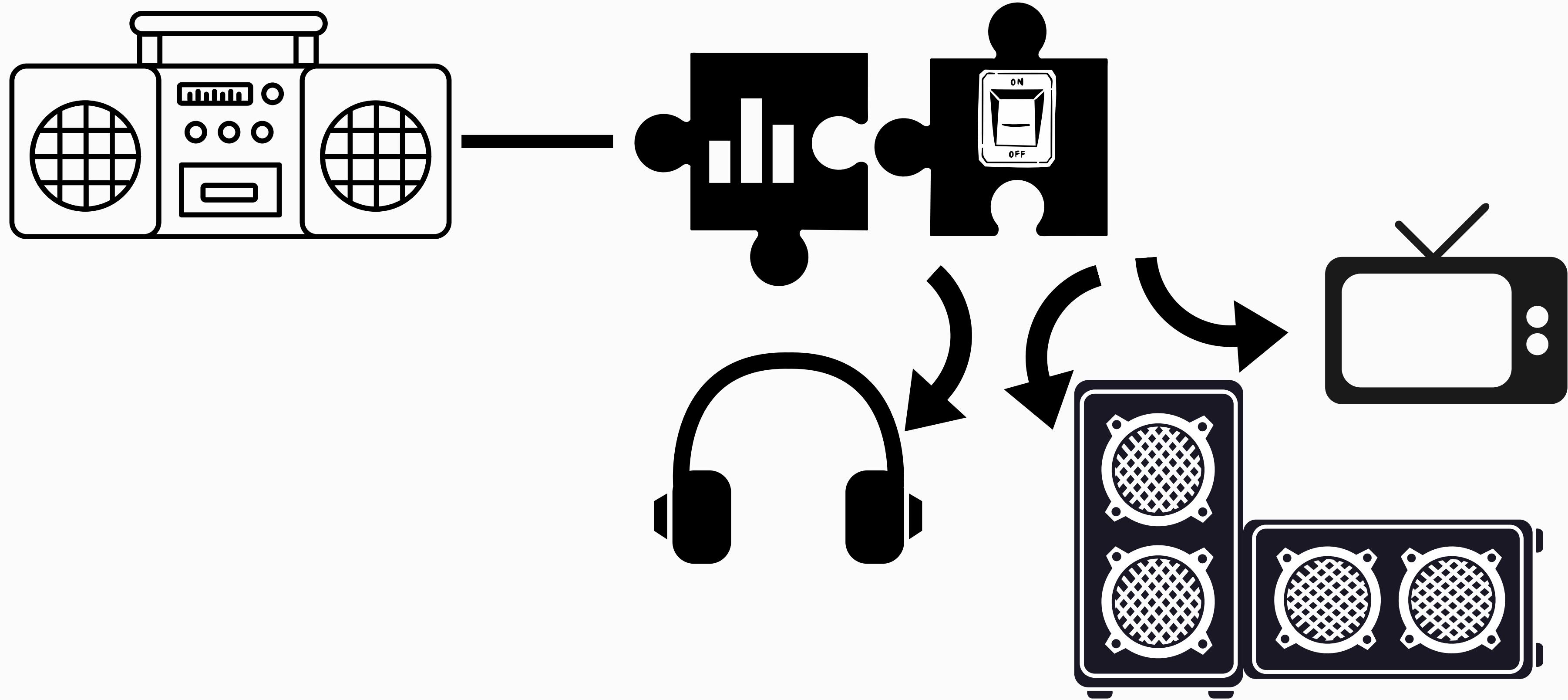
Implementation with design pattern

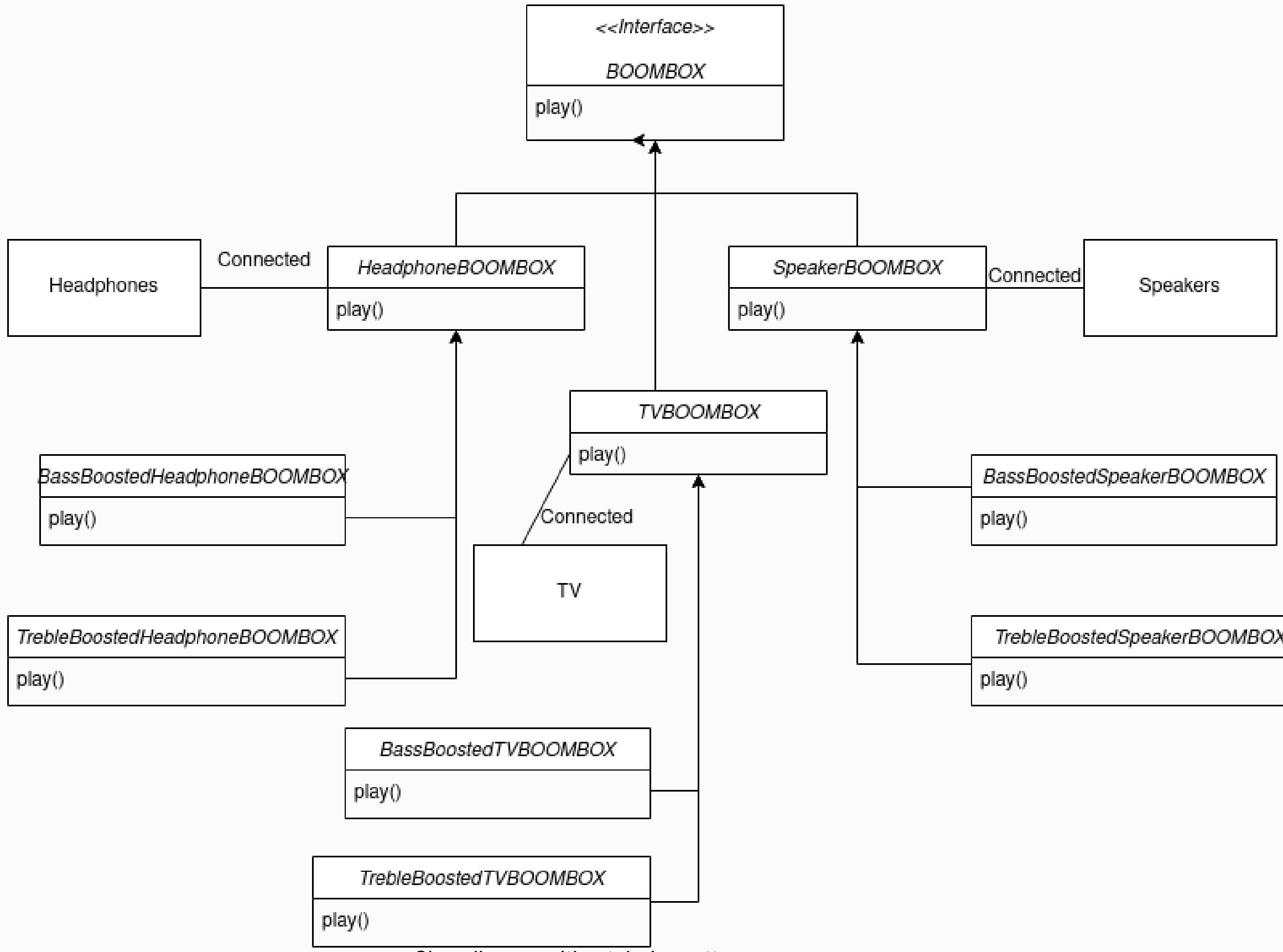
Strategy and Decorator

Patterns are very
powerful when
used together

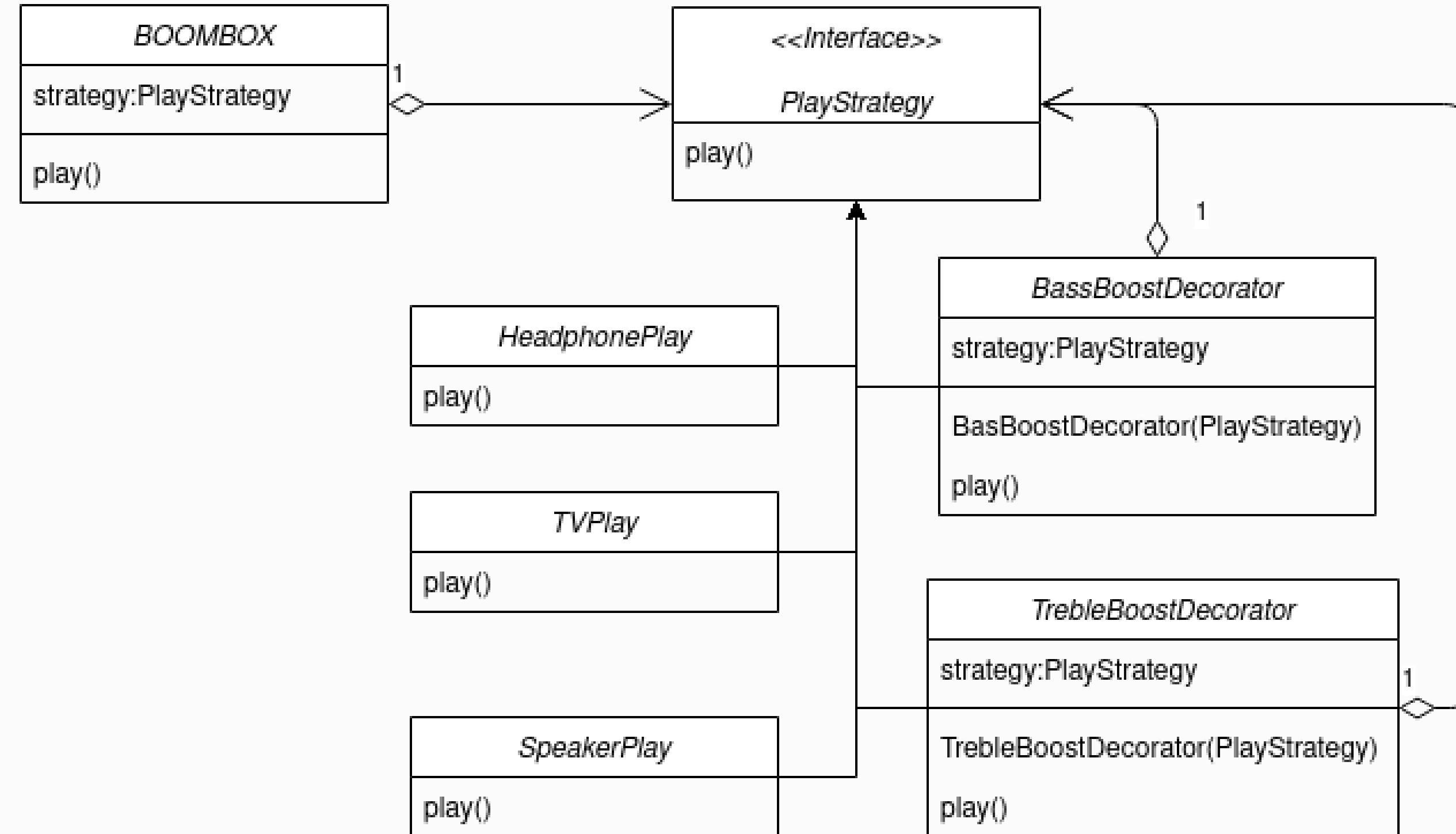




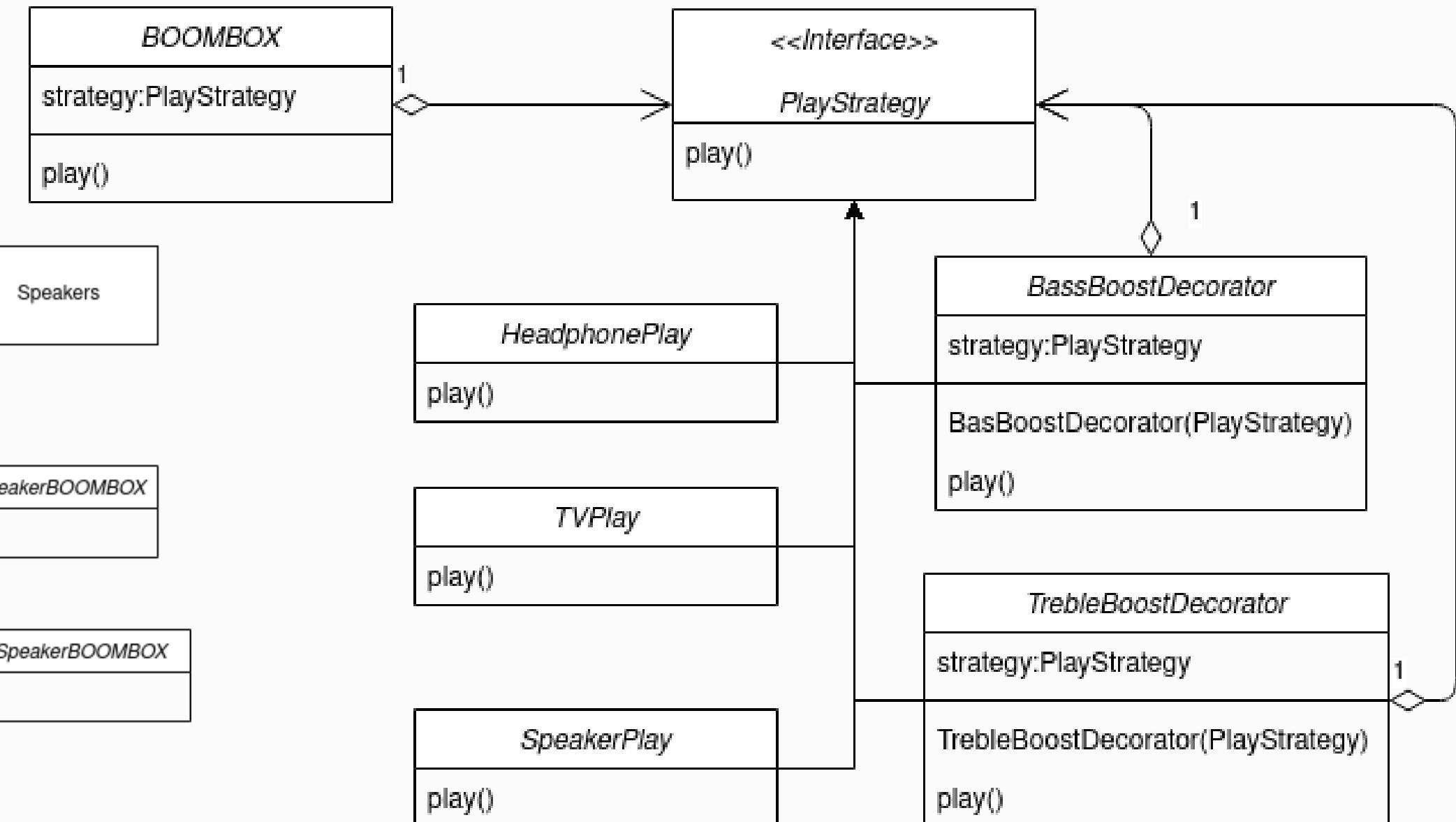
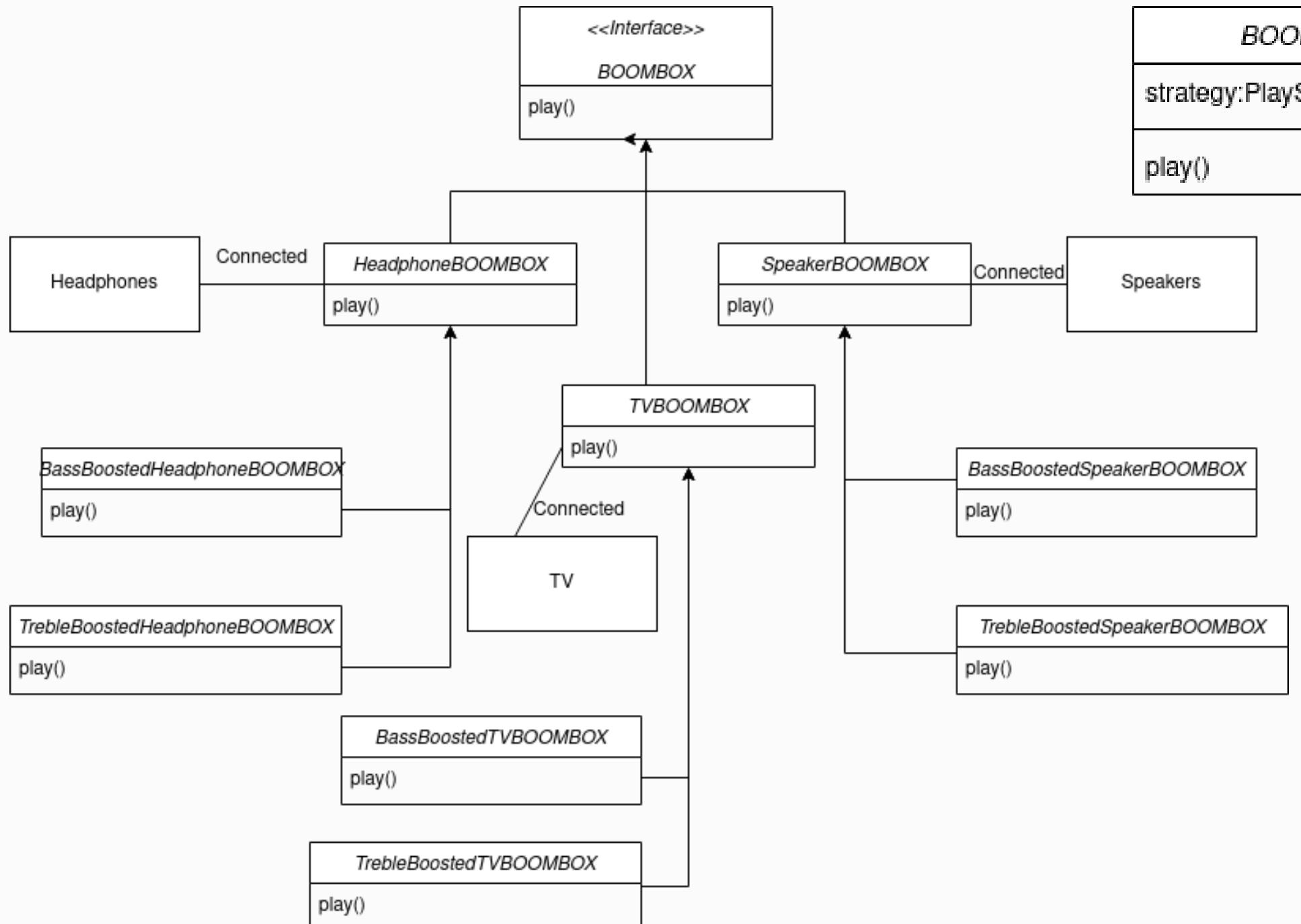


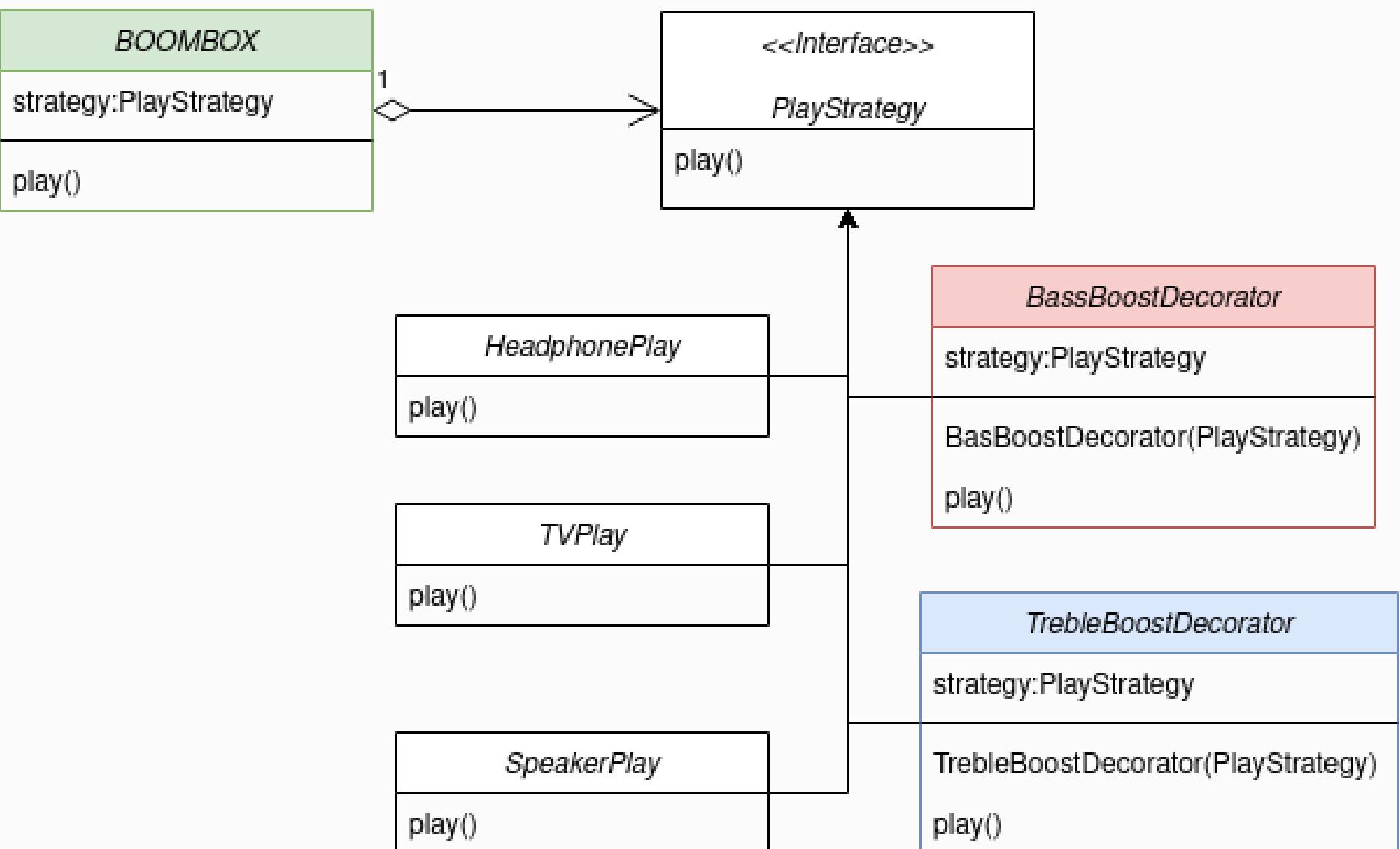
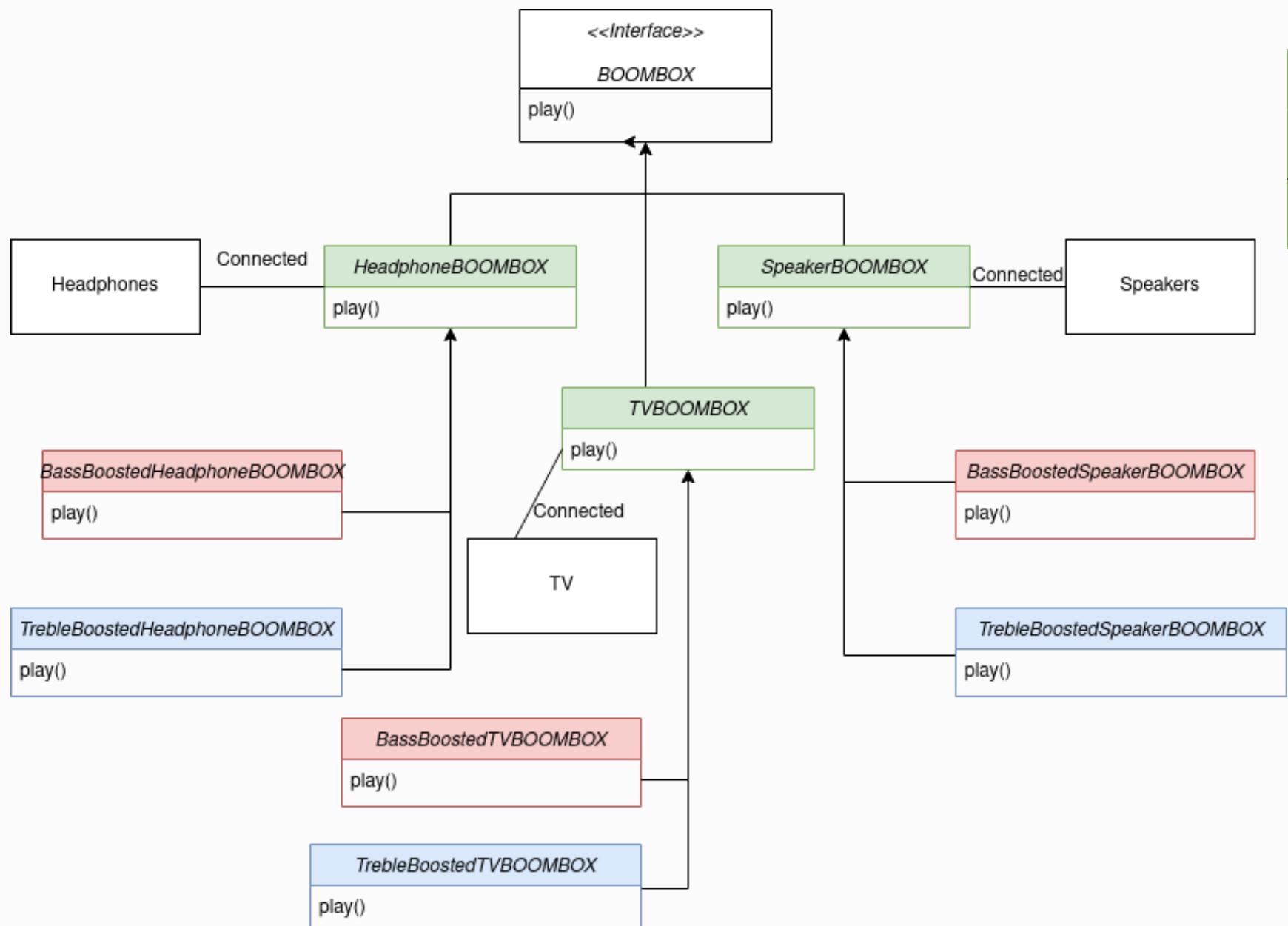


Class diagram without design patterns



Class diagram with design patterns



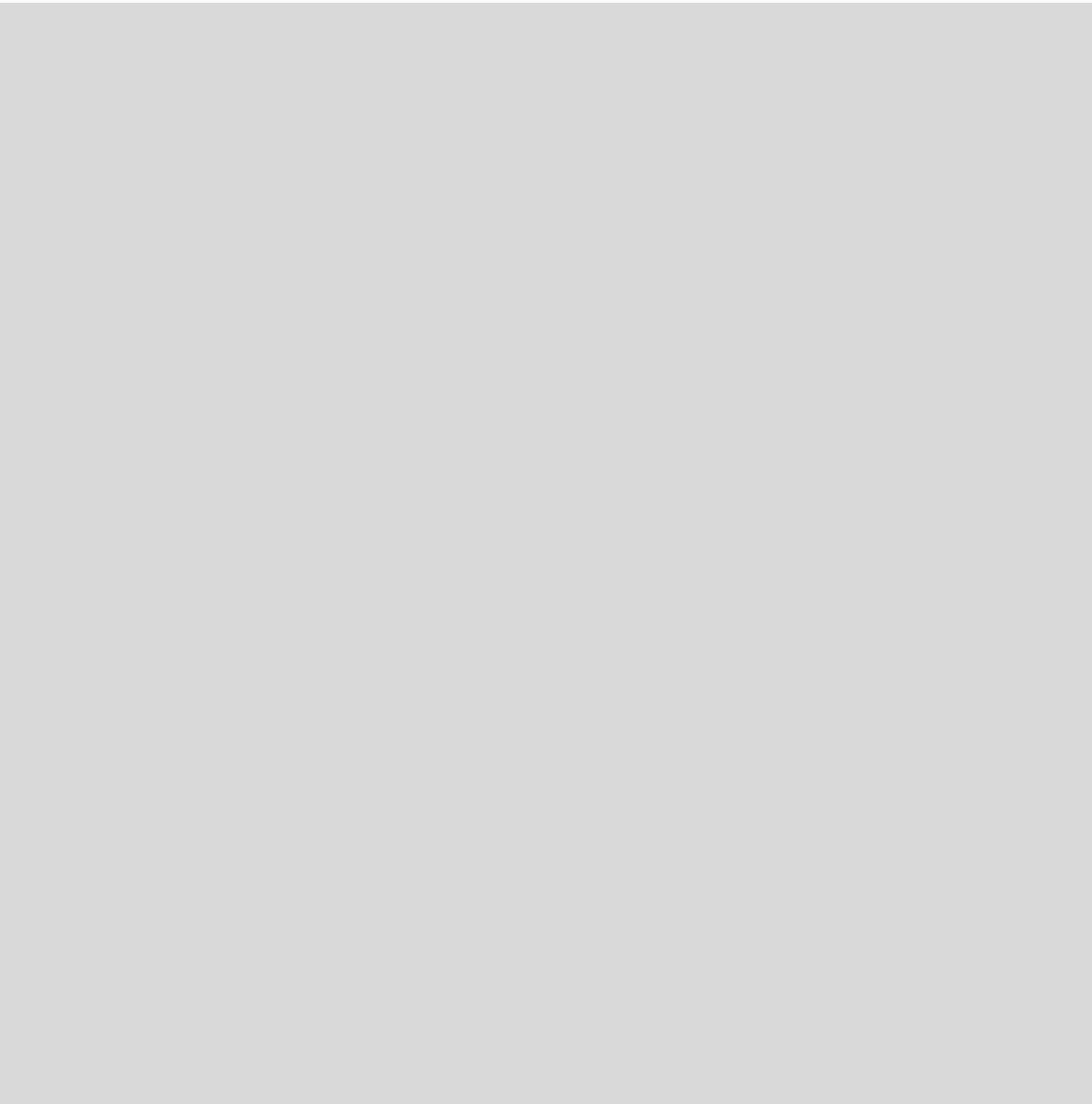


REFERENCES

- [1] ELİSA BANIASSAD, IVAN BESCHASTNİKH, REİD HOLMES, GREGOR KİCZALES, AND MEGHAN ALLEN. 2019. LEARNING TO LISTEN FOR DESIGN. IN PROCEEDİNGS OF THE 2019 ACM SIGPLAN INTERNATIONAL SYMPOSIUM ON NEW IDEAS, NEW PARADİGMS, AND REFLECTIONS ON PROGRAMMING AND SOFTWARE (ONWARD! 2019). ASSOCIATION FOR COMPUTİNG MACHİNERY, NEW YORK, NY, USA, 179–186. DOI:[HTTPS://DOI.ORG/10.1145/3359591.3359738](https://doi.org/10.1145/3359591.3359738)



(0,0)



fill(r,g,b)

rect(x,y,width,height)

(width,height)

NEW ASSIGNMENT

Create two enemies,

- Bouncing Enemy: Inverses the speed when hits a side
- Random Enemy: Randomizes the speed when hits a side

Both looking the same a blue square with side

NEW ASSIGNMENT

Create an extension of enemies

- Heavy Enemies: affected from gravity
(ySpeed increases over time)

