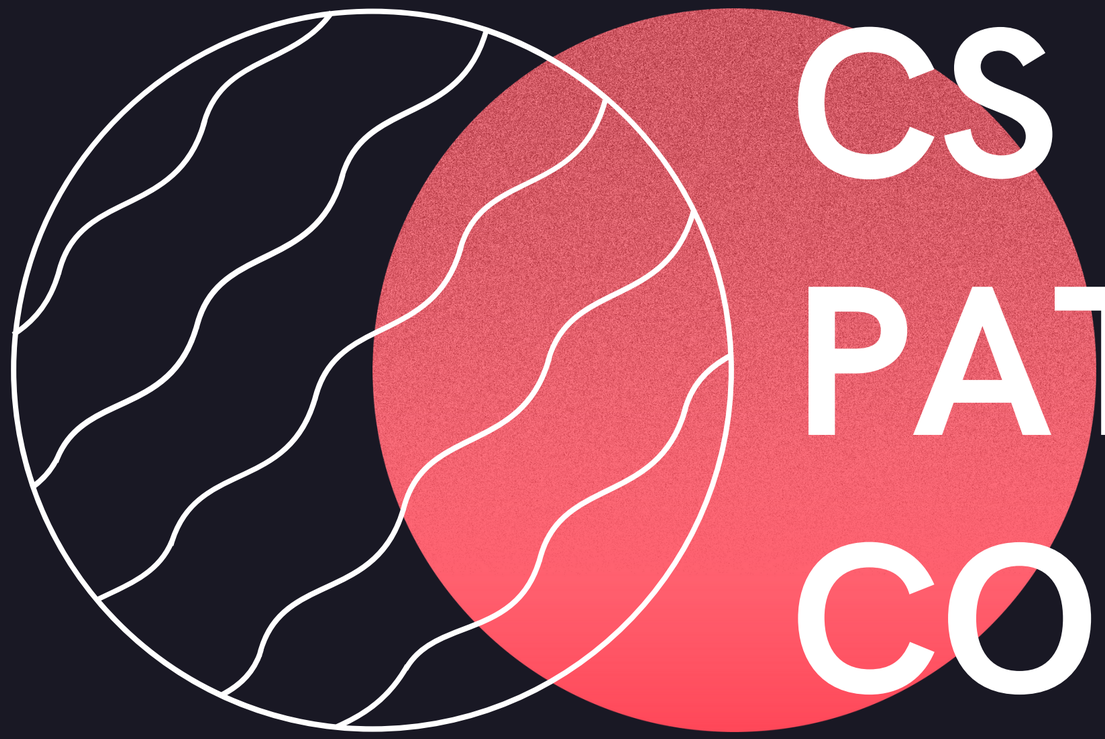
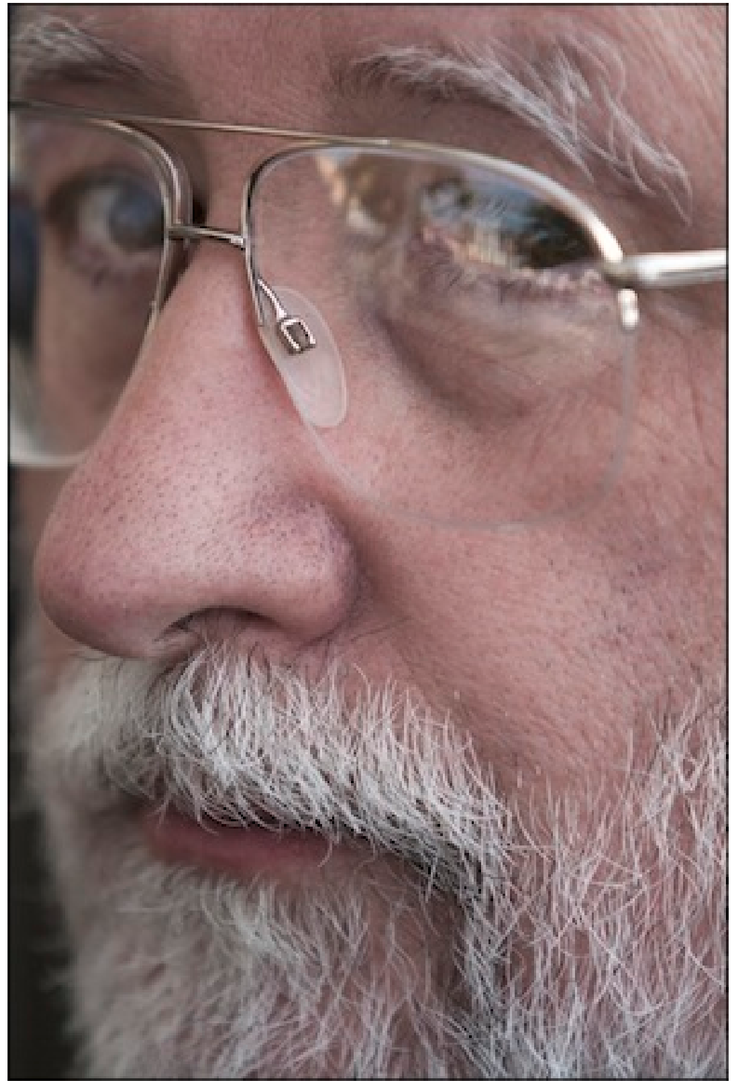


CS 319 DESIGN PATTERNS AND CODE SMELLS

Ata Yurtsever

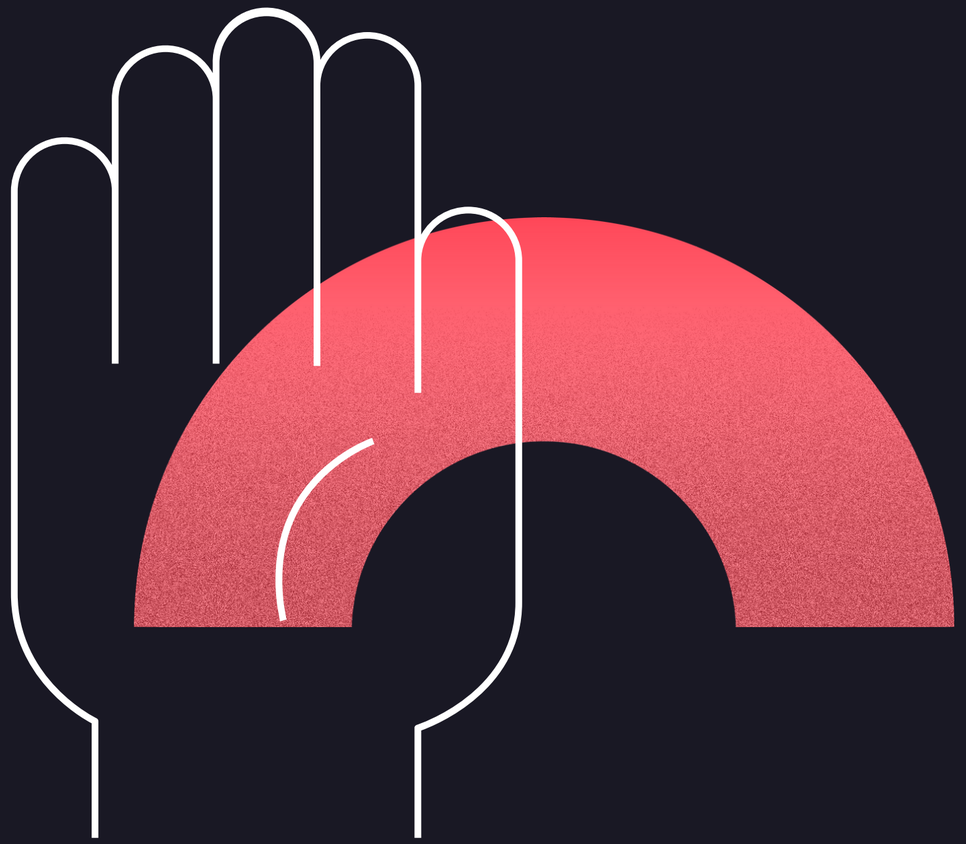
GABRIEL IMPLIES THAT CODE SPEAKS TO THE PROGRAMMER THE SAME WAY A POEM SPEAKS TO THE POET. JUST AS ARTISTS LOOK FOR ASYMMETRY, IMBALANCE, OR LACK OF RHYTHM, DEVELOPERS LOOK FOR DEFICIENCIES IN THEIR CODE.

[1]



RICHARD P. GABRIEL

[1]Elisa Baniassad, Ivan Beschastnikh, Reid Holmes, Gregor Kiczales, and Meghan Allen. 2019. Learning to listen for design. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019). Association for Computing Machinery, New York, NY, USA, 179–186. DOI:https://doi.org/10.1145/3359591.3359738

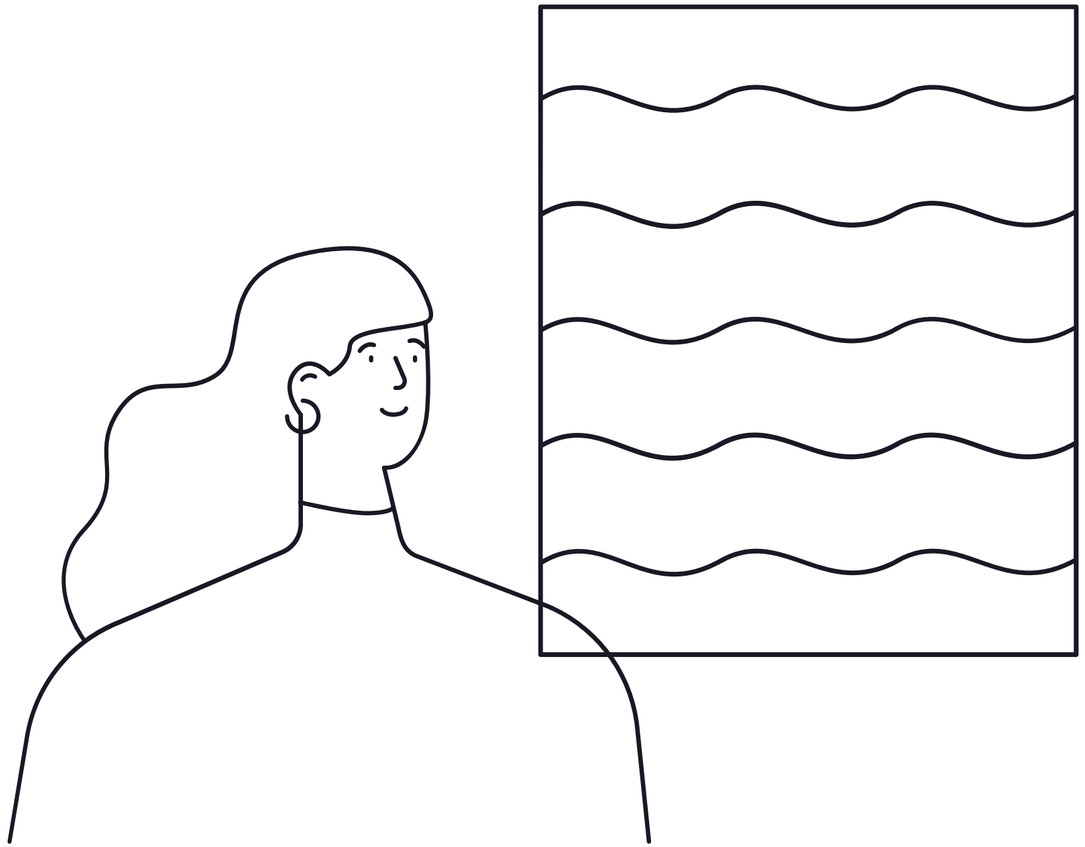


WHAT TO LISTEN FOR

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.

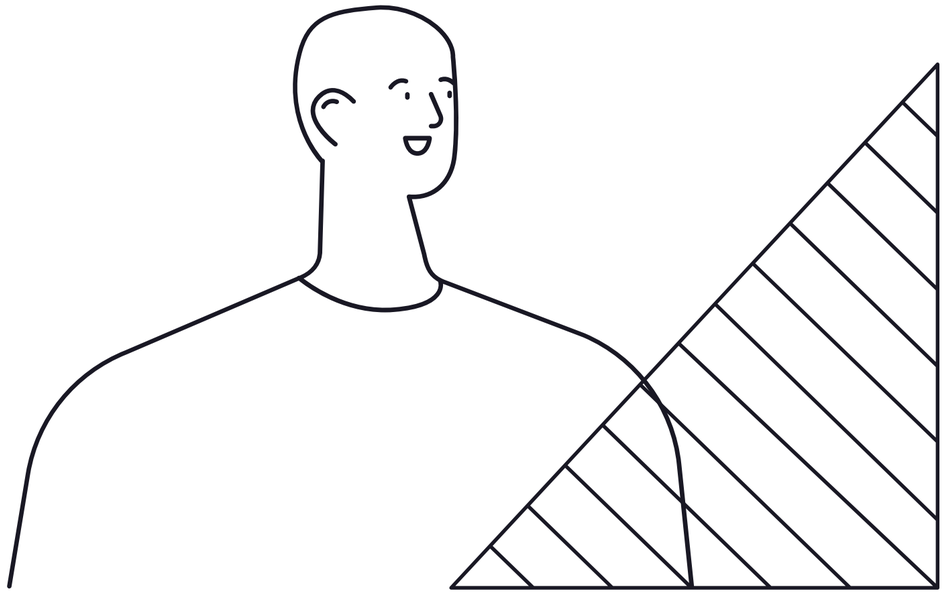
[1]



WHAT DOES IT SAY?

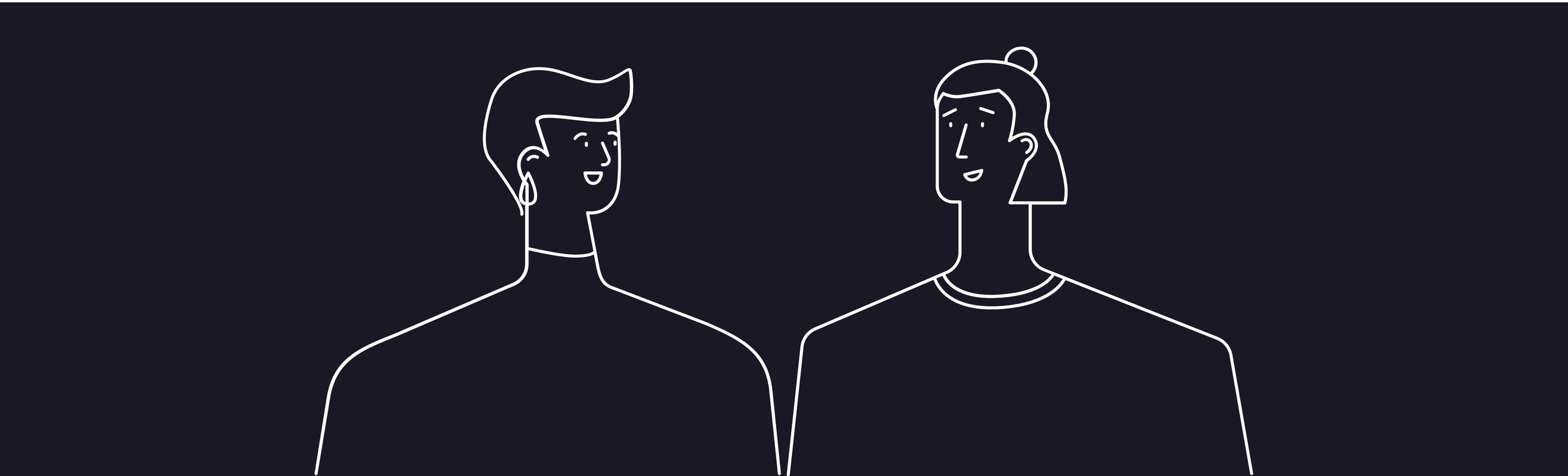
…with this wisdom and arsenal, programmers listen to the code, and they interpret what the code is saying by abstracting its ‘words’ into phrases and concepts which are, in fact, design principle violations.
[1]

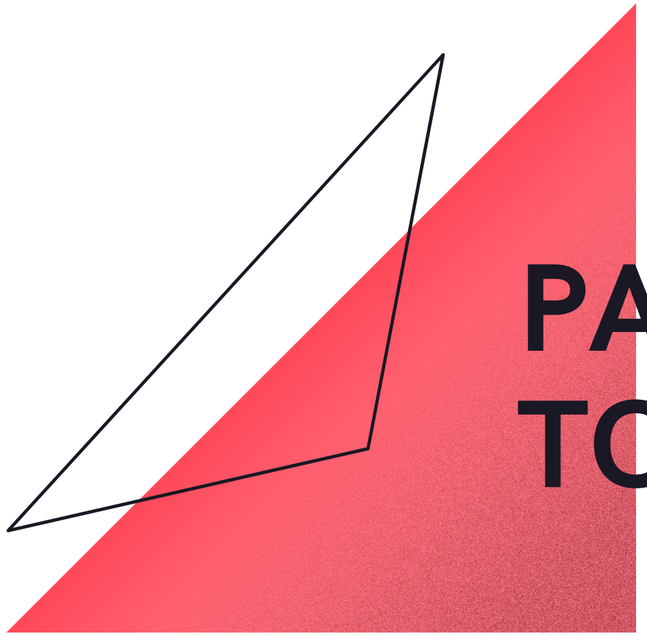
Solving these changes require a good understanding of how designs are implemented. You cannot quench someones thirst without knowing what water is.



WHAT ARE THESE DESIGN PATTERNS

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





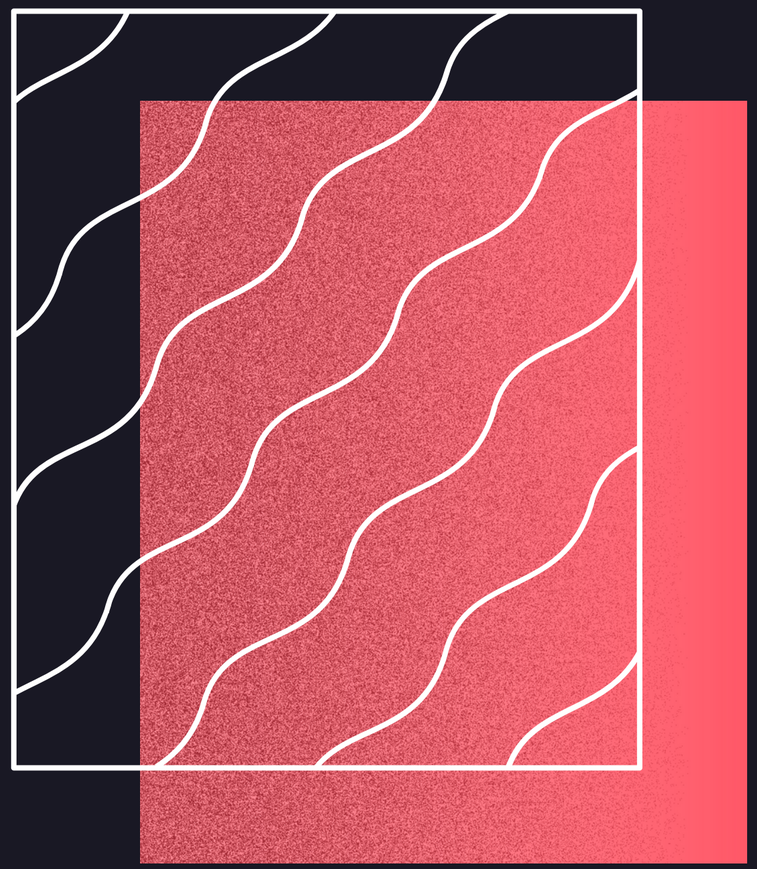
PATTERNS OF TODAY'S LECTURE

STRATEGY PATTERN

ENCAPSULATING ALGORITHMS FOR CODE REUSE

DECORATOR PATTERN

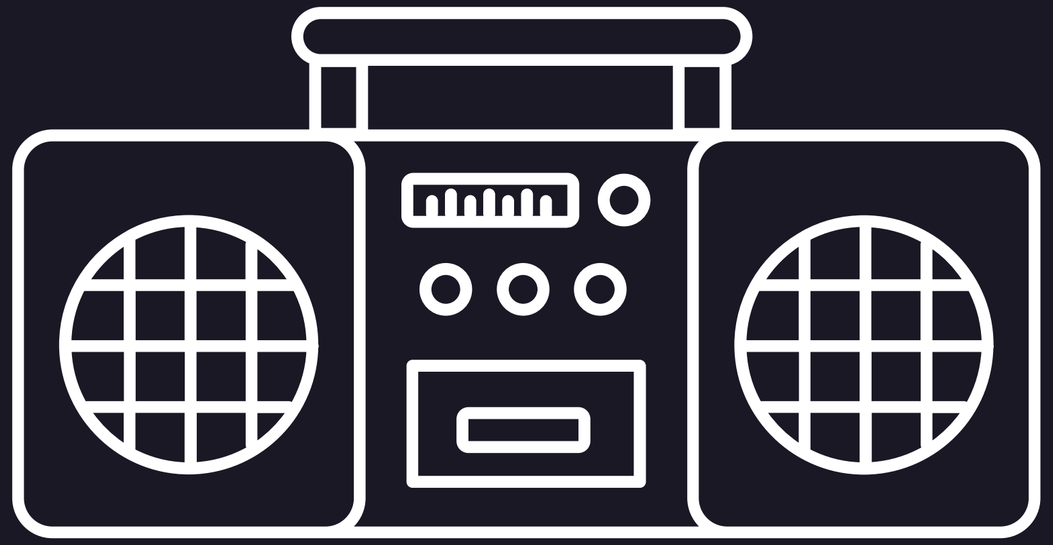
EXTENDING ALGORITHMS FOR CODE REUSE

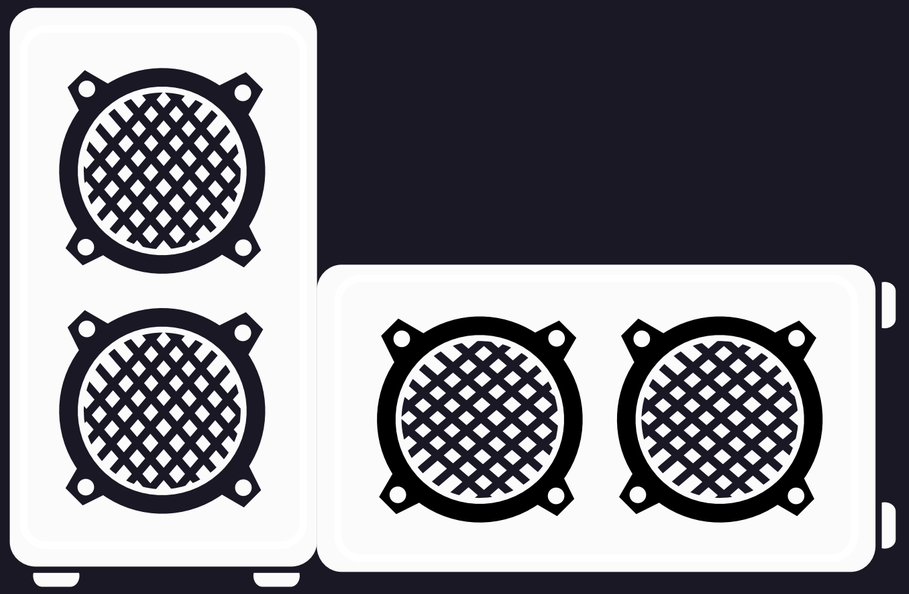


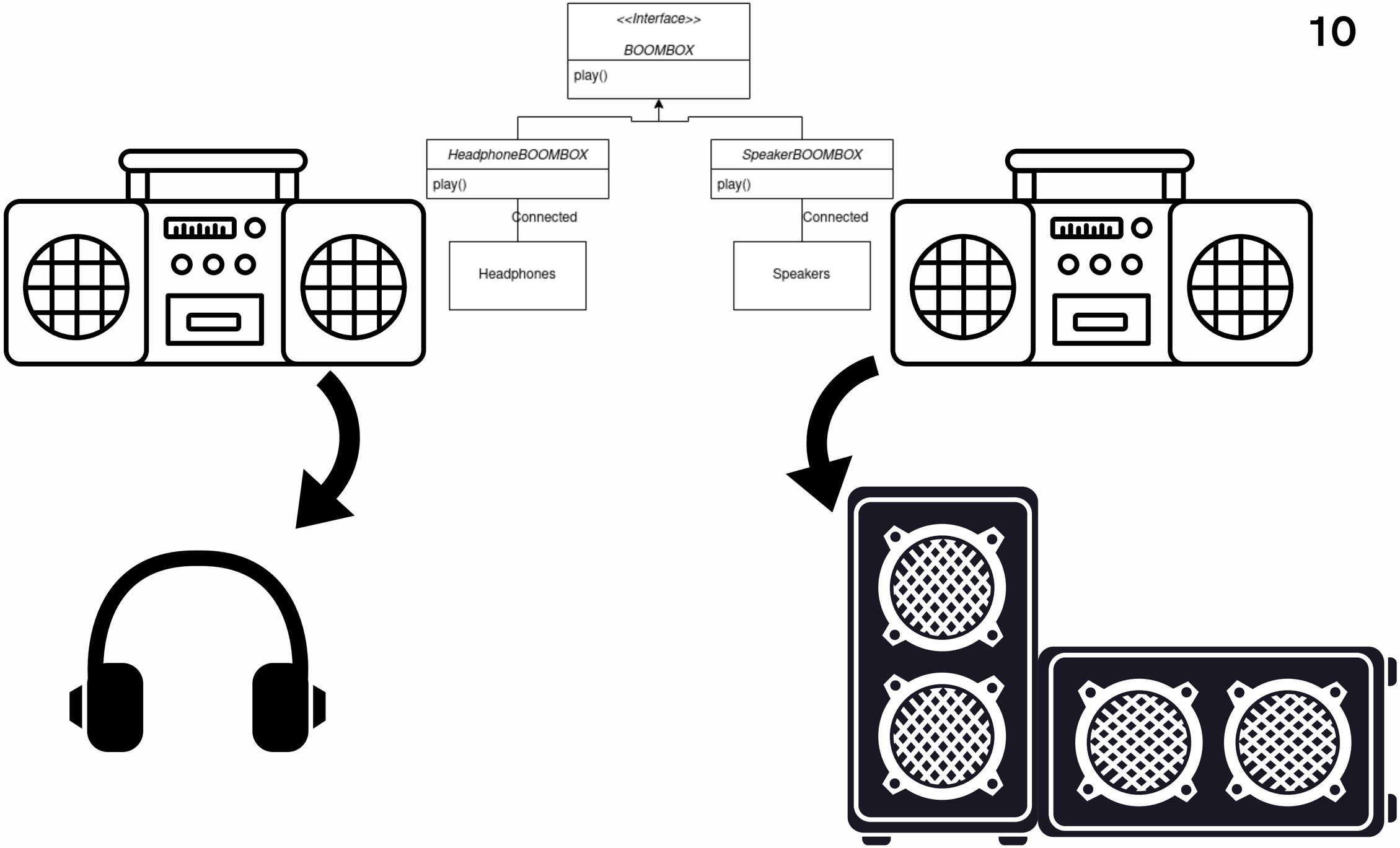
STRATEGY PATTERN

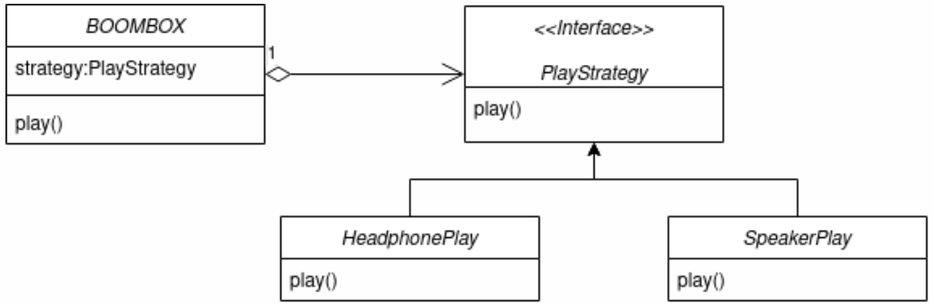
Strategy pattern encapsulates algorithms and so that you can choose algorithm in runtime. It also allows code reuse by allowing multiple agents with similar properties differentianting from implementions to be grouped under the same tent.

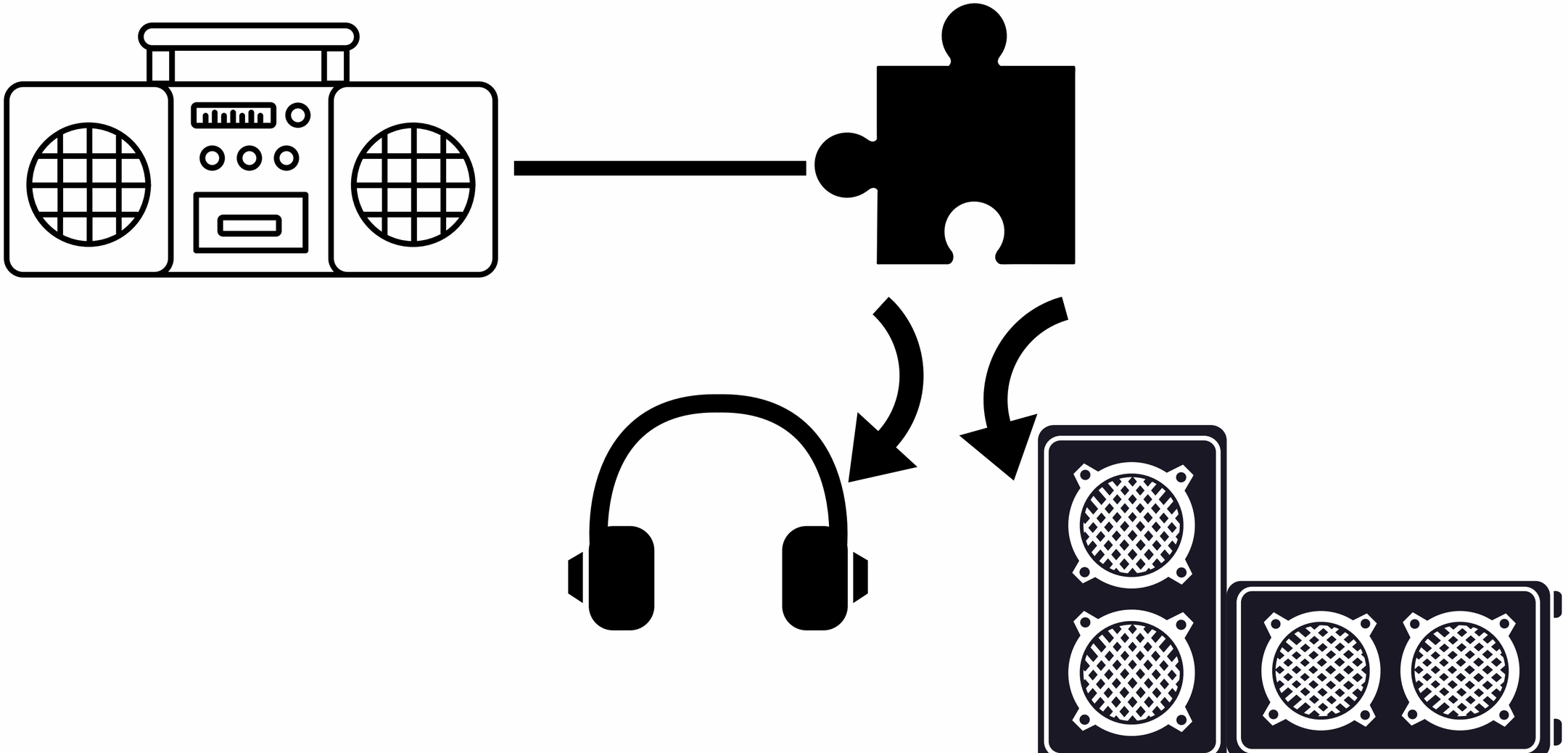
A good example can be the music player system.

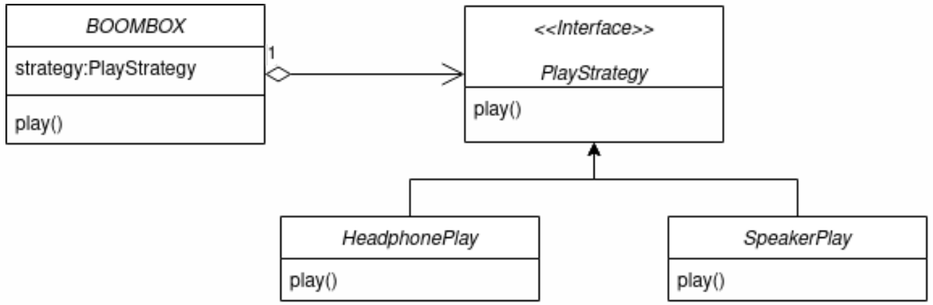


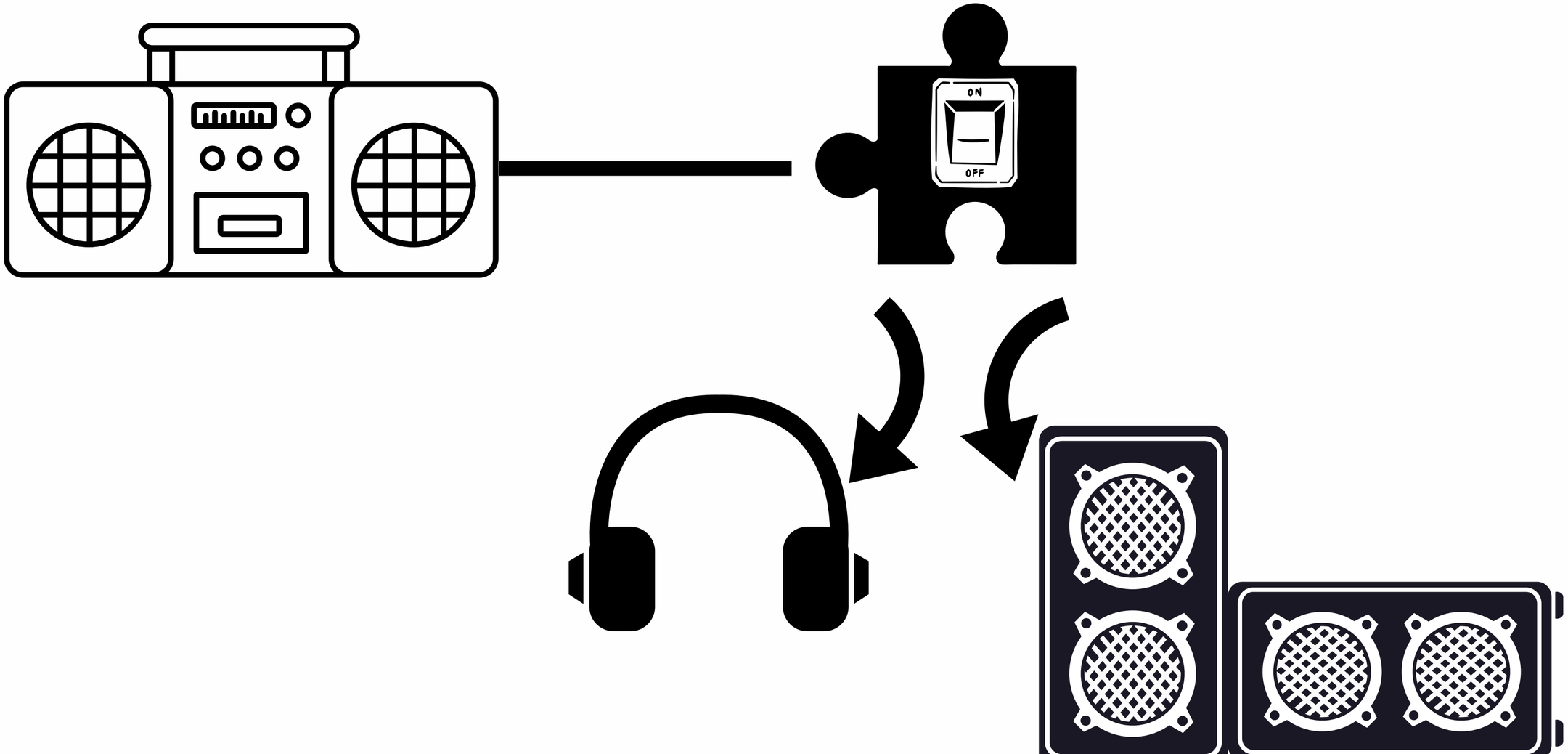



<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
SpeakerBOOMBOX
play()
Connected
Speakers


BOOMBOX
strategy:PlayStrategy
play()
1
<<Interface>>
PlayStrategy
play()
HeadphonePlay
play()
SpeakerPlay
play()




BOOMBOX
strategy:PlayStrategy
play()
1
<<Interface>>
PlayStrategy
play()
HeadphonePlay
play()
SpeakerPlay
play()

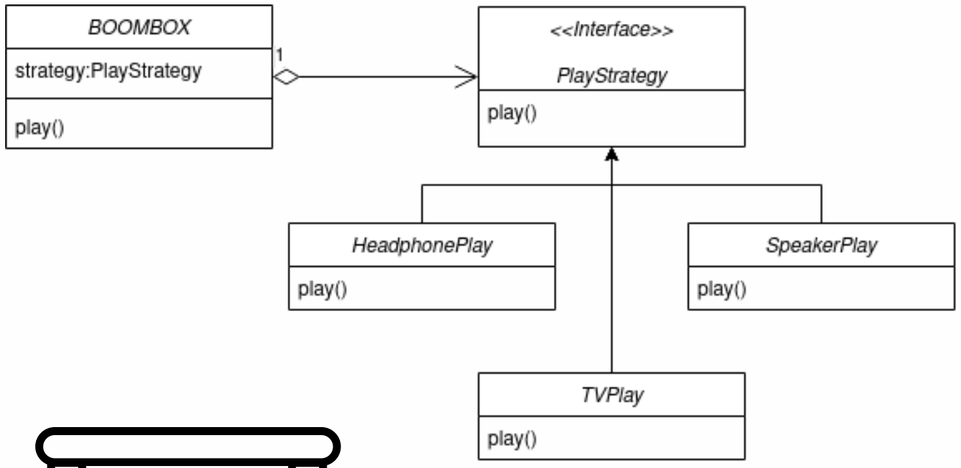

ON
OFF

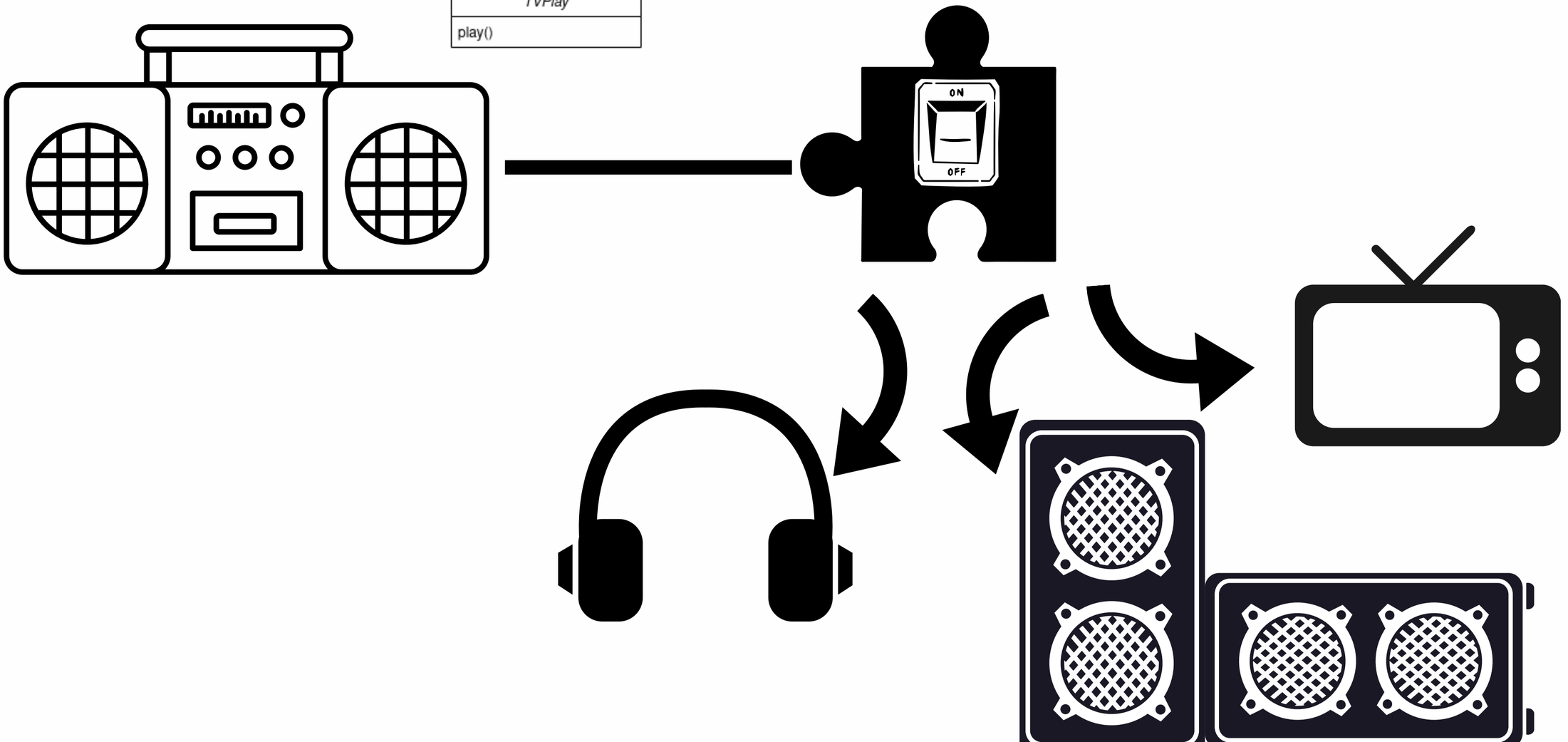
```java
public interface Boombox {
    SoundSignal music;
    void play();
}

public class HeadphoneBoombox extends Boombox{
    void play(){
        //I am playing this on headphones
        Headphones.play(this.music)
    }
}
```

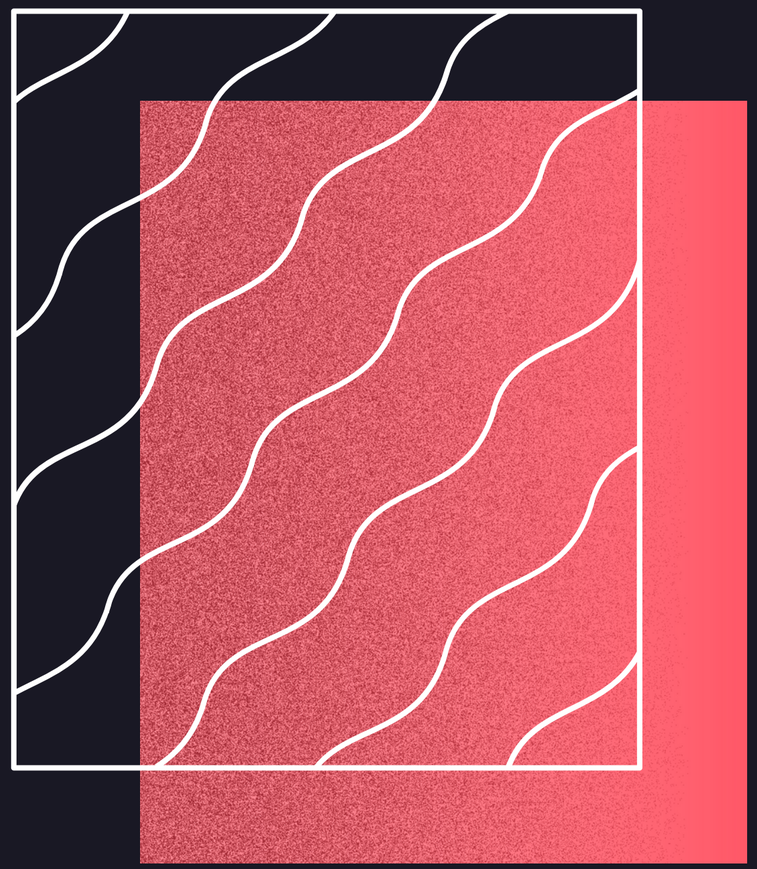
without design patterns

```java
public interface PlayStrategy{
    void play(SoundSignal);
}

public class Boombox{
    SoundSignal music;
    PlayStrategy playStrategy;
    public Boombox(PlayStrategy playStrategy){
        this.playStrategy = playStrategy;
    }
    void play(){
        playStrategy(music);
    }
}

public class HeadphonePlayStrategy implements PlayStrategy{
    //you can get the required stuff as parameters
    void play(SoundSignal music){
        //play on headphones
        Headphone.play(music);
    }
}
```

with design patterns


BOOMBOX
strategy:PlayStrategy
play()
1
<<Interface>>
PlayStrategy
play()
HeadphonePlay
play()
SpeakerPlay
play()
TVPlay
play()

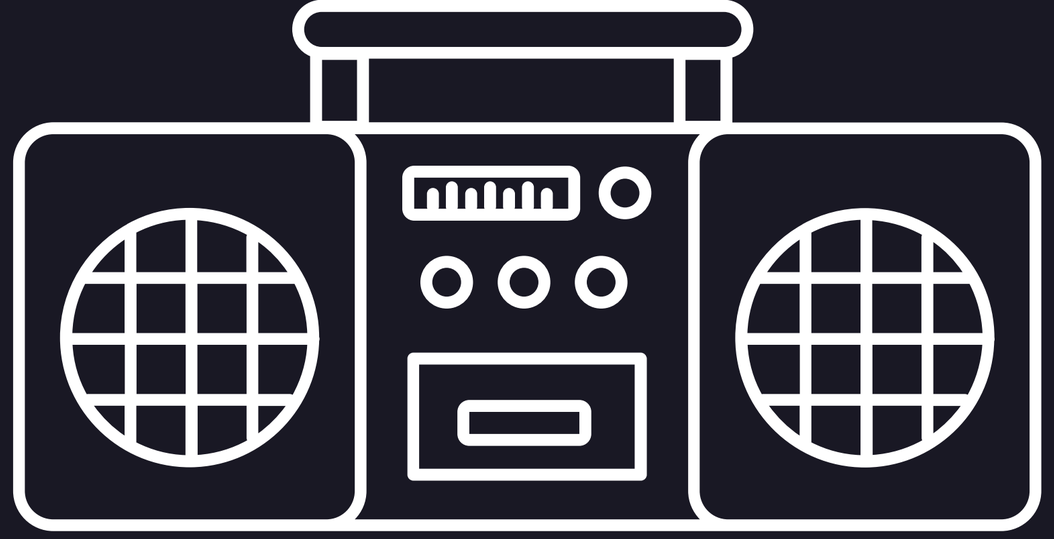

ON
OFF



DECORATOR PATTERN

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

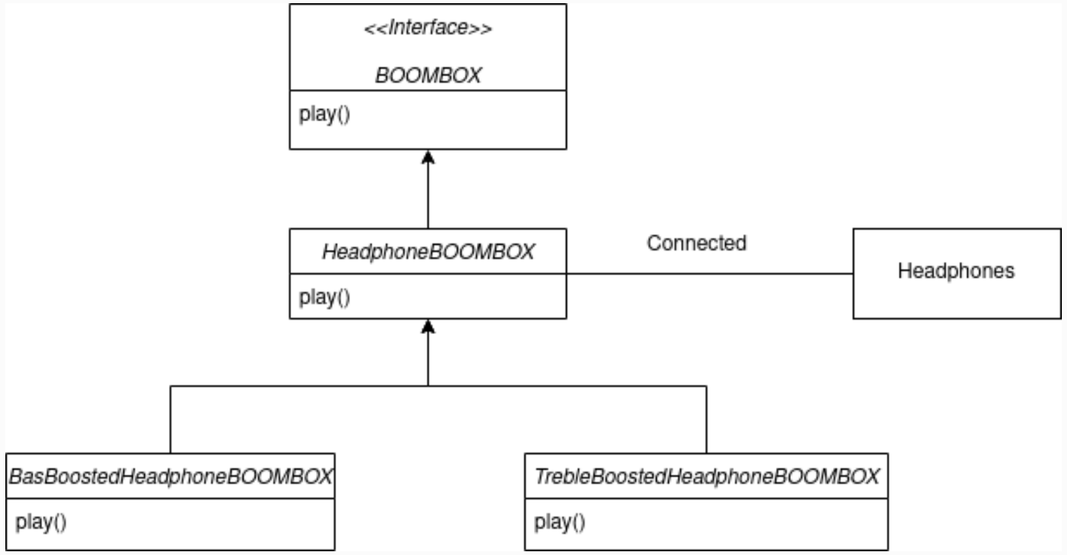
A good example can be again our music player system.

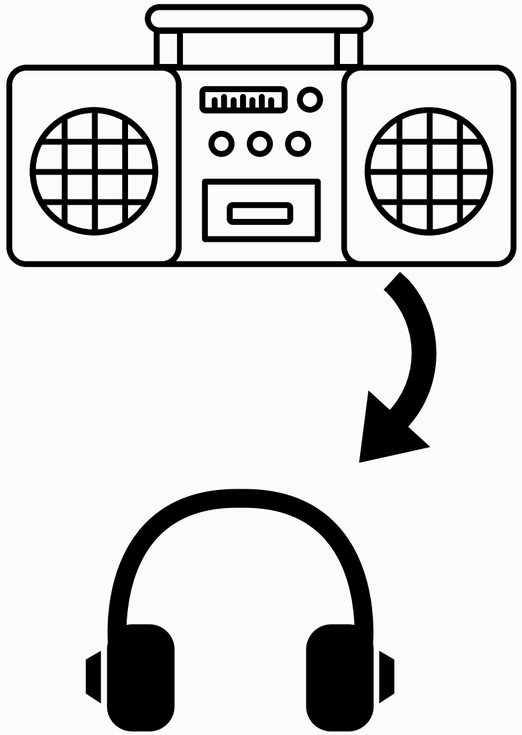


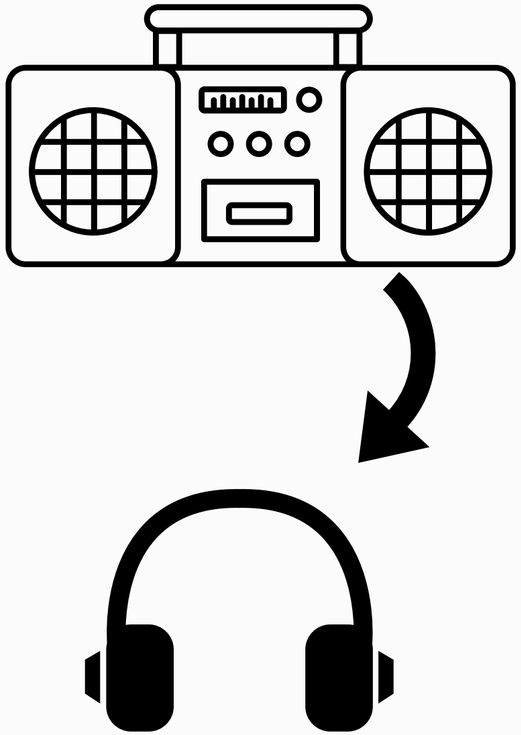
BASS BOOSTED

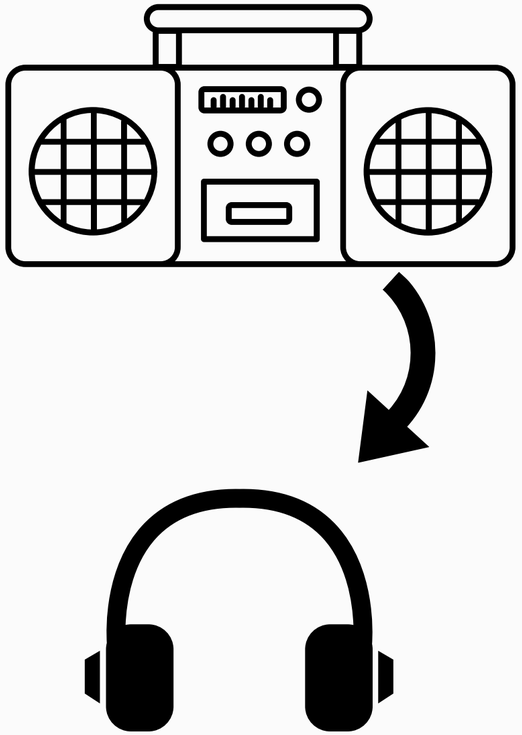
NORMAL

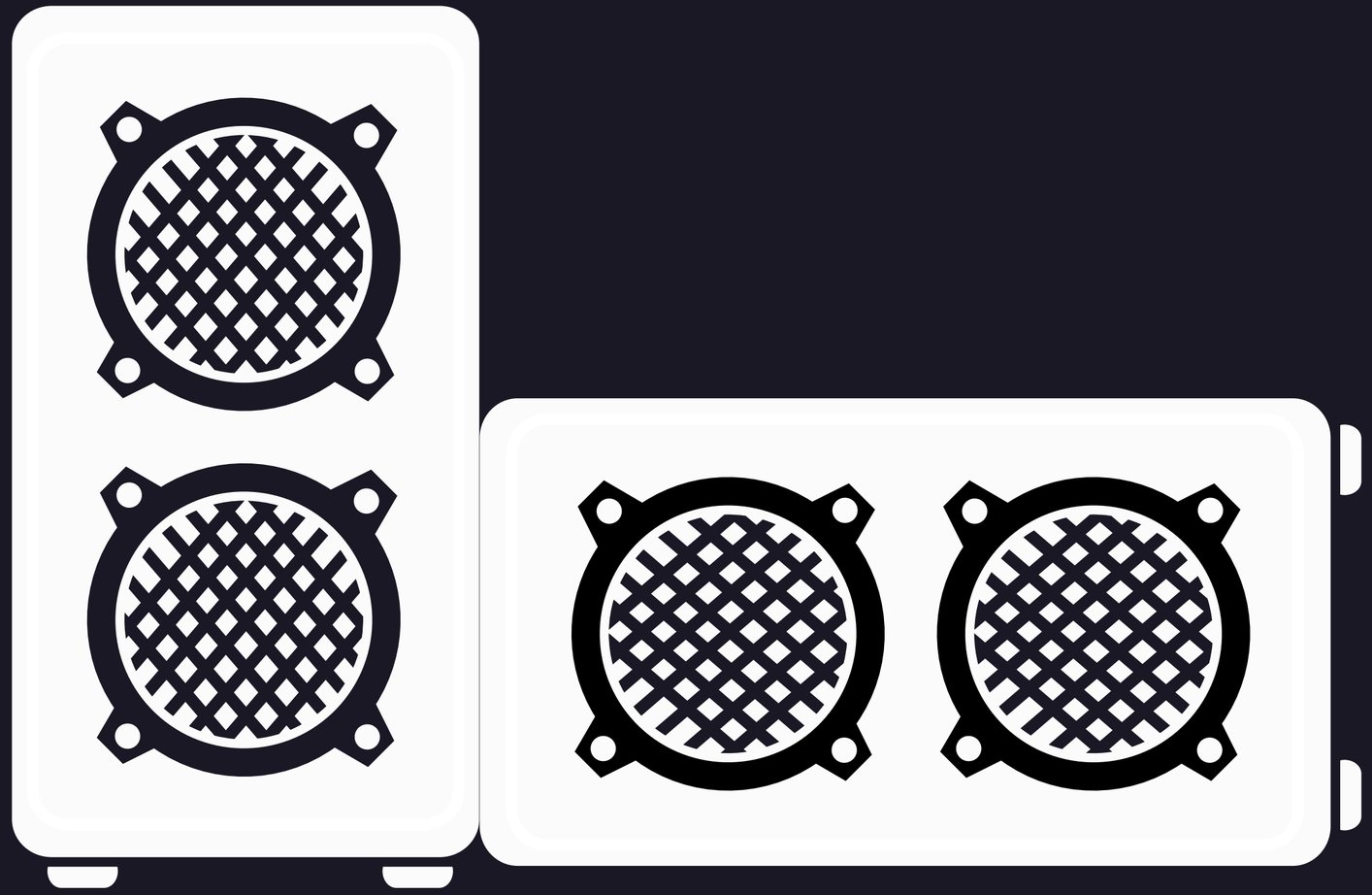
TREBLE BOOSTED

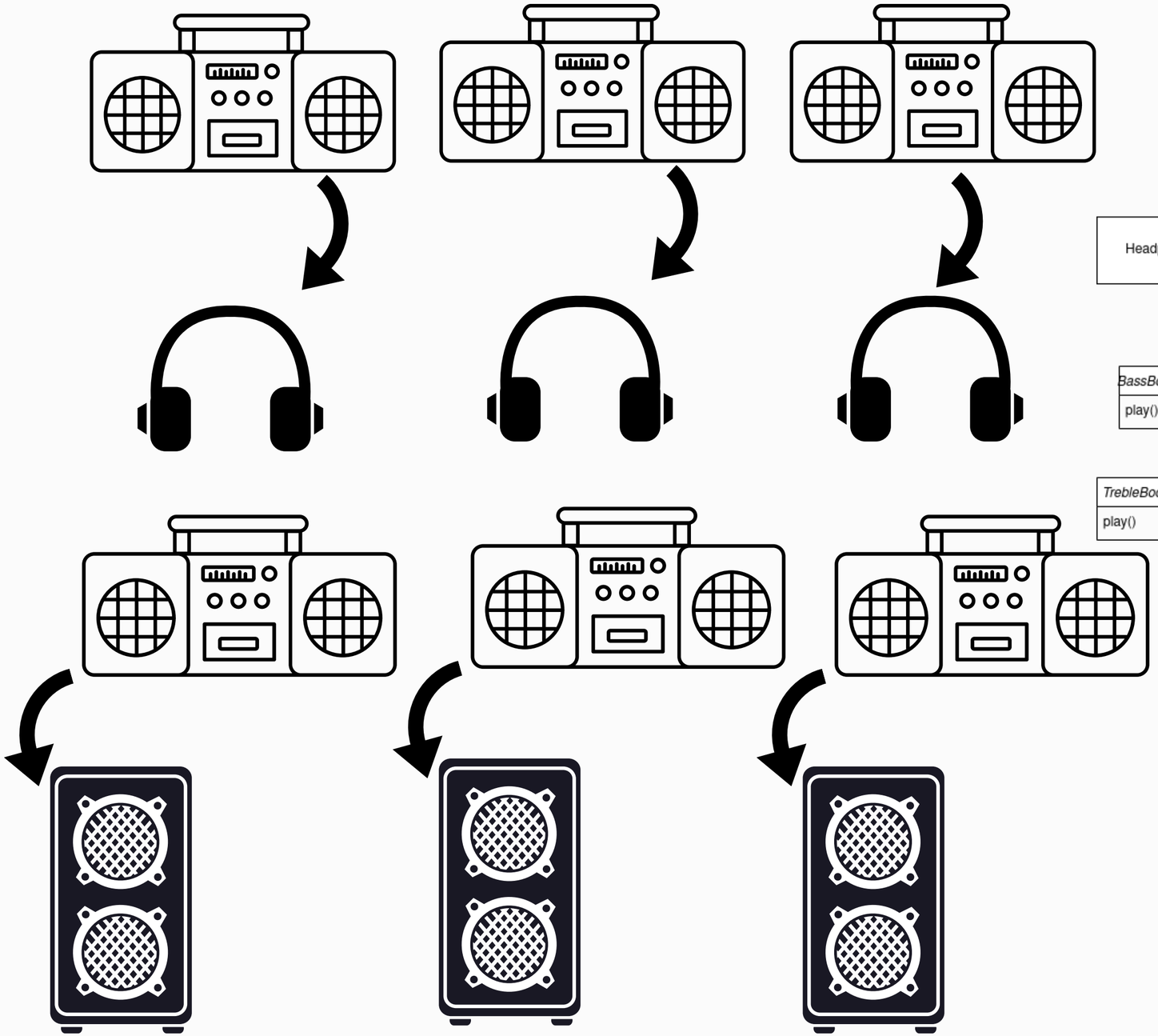

<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
BasBoostedHeadphoneBOOMBOX
play()
TrebleBoostedHeadphoneBOOMBOX
play()

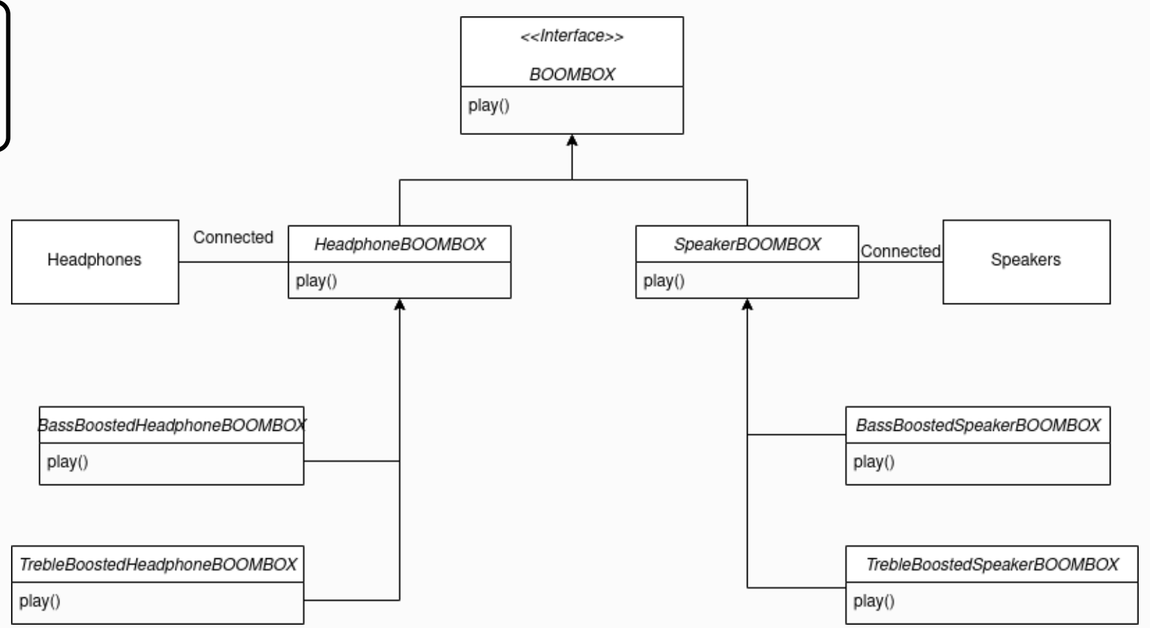


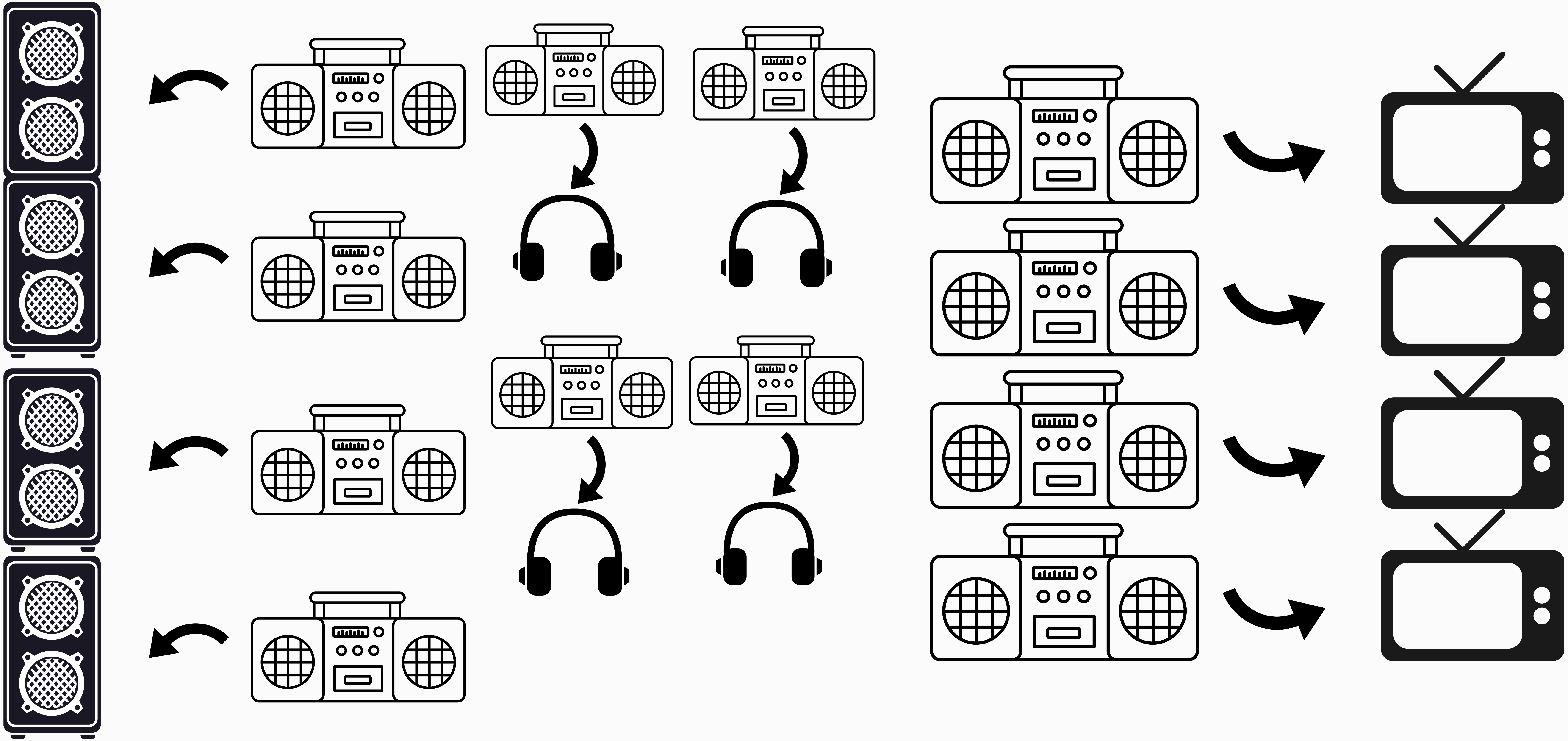






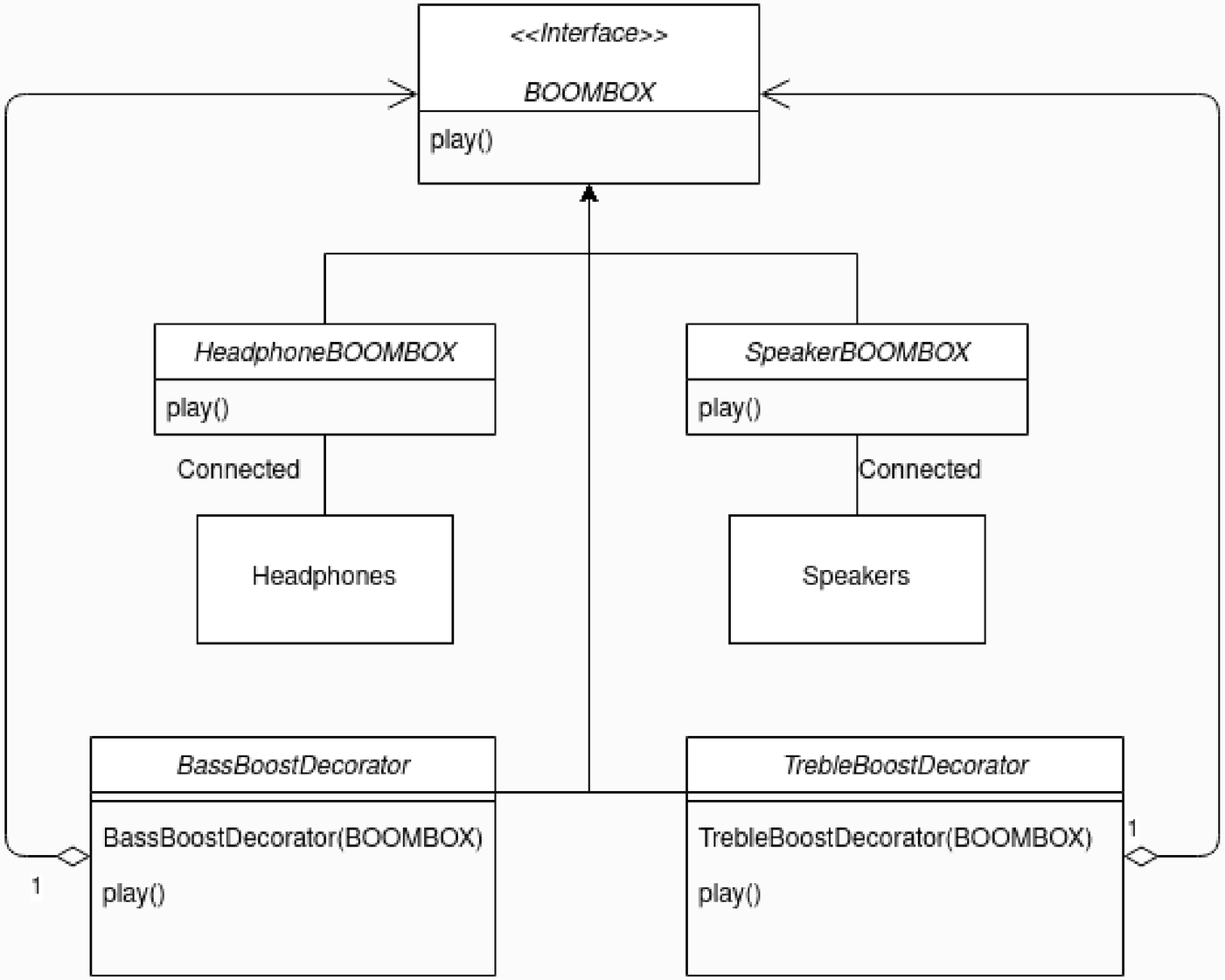


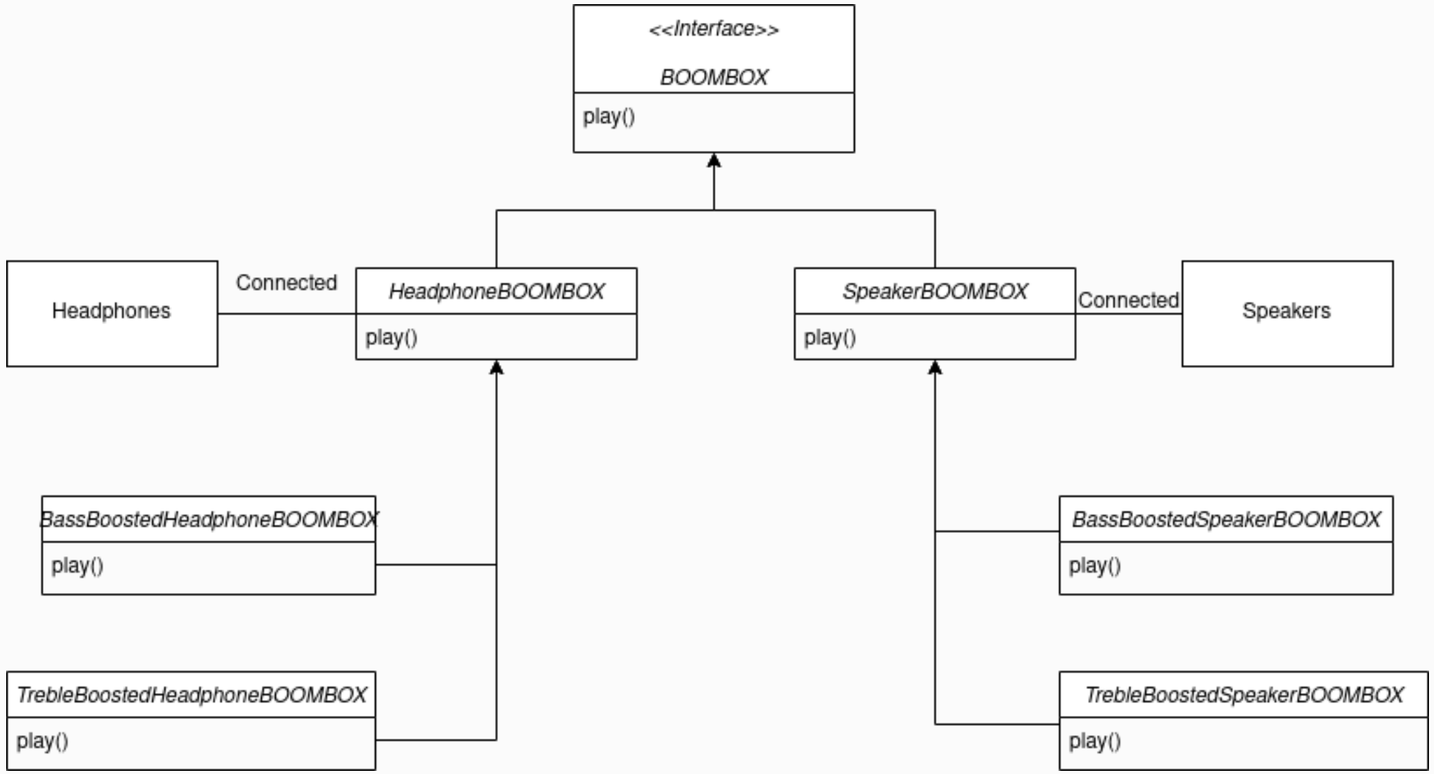

<<Interface>>
BOOMBOX
play()
Headphones
Connected
HeadphoneBOOMBOX
play()
SpeakerBOOMBOX
play()
Connected
Speakers
BassBoostedHeadphoneBOOMBOX
play()
TrebleBoostedHeadphoneBOOMBOX
play()
BassBoostedSpeakerBOOMBOX
play()
TrebleBoostedSpeakerBOOMBOX
play()



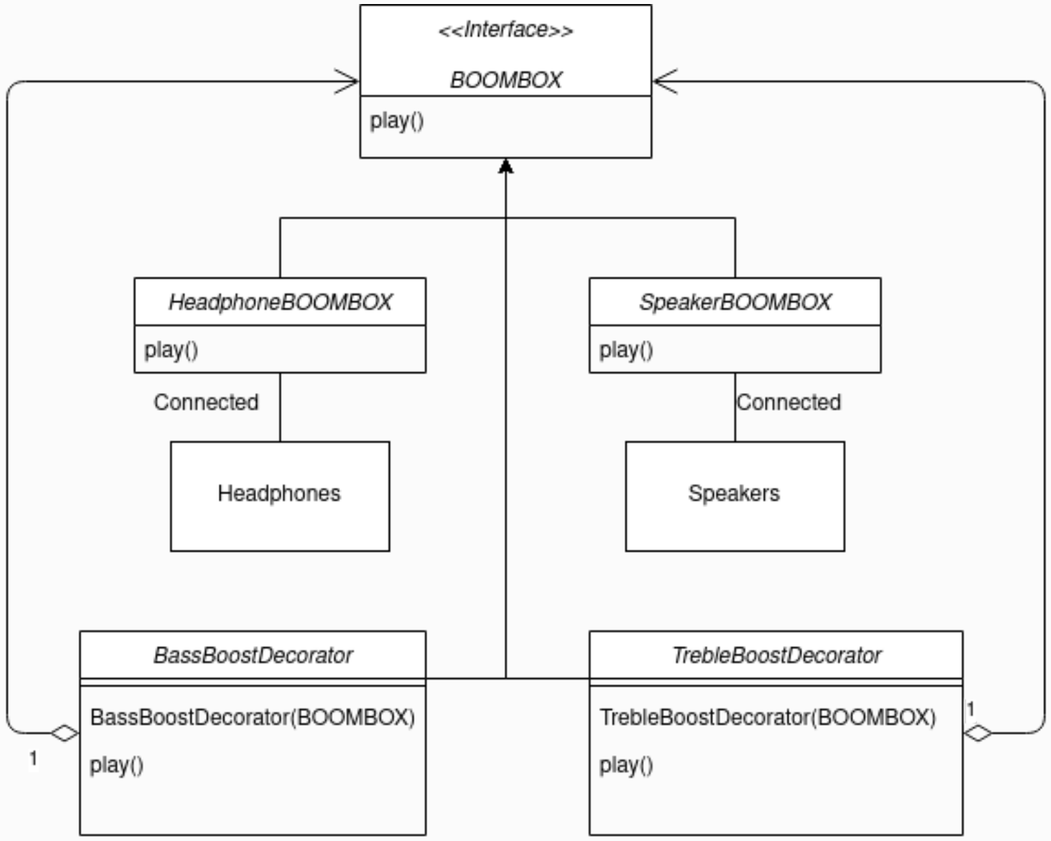
SO WHAT SHOULD WE DO?

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

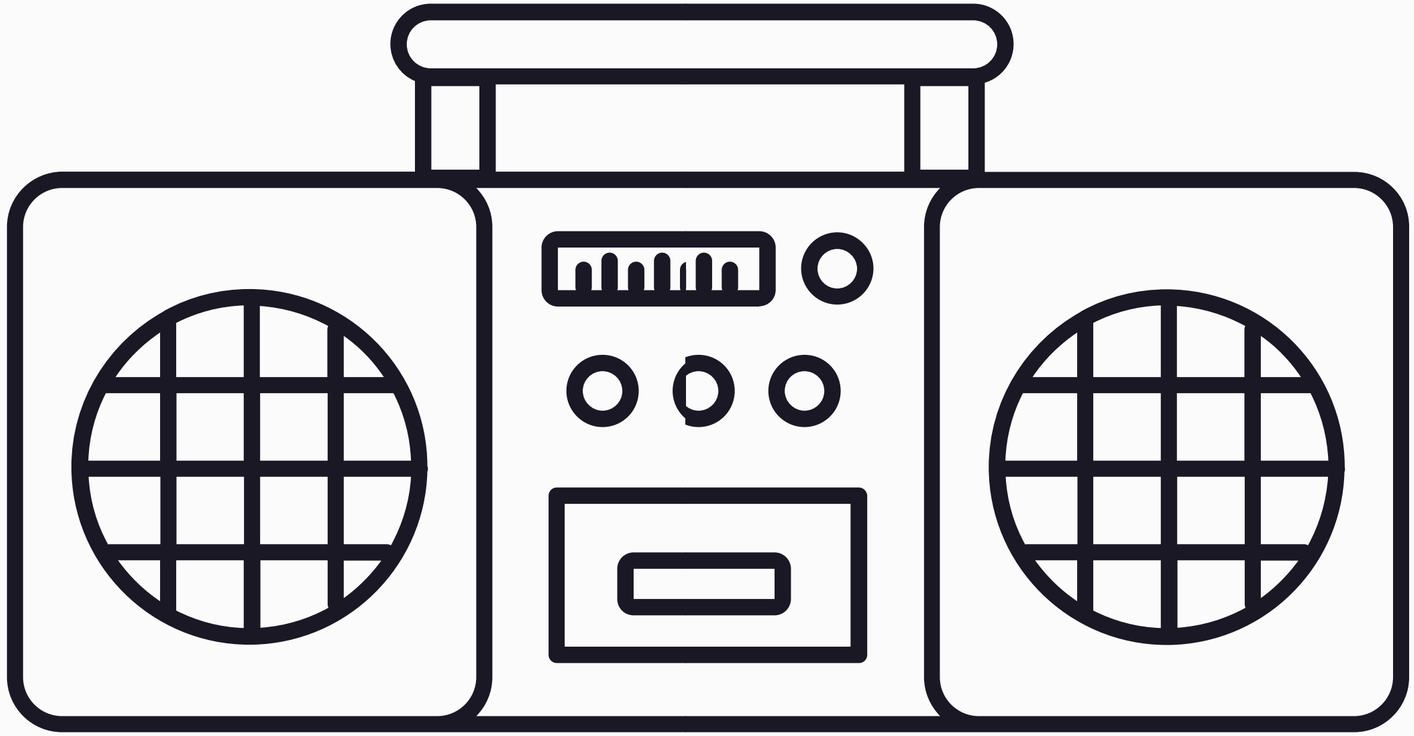

<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
SpeakerBOOMBOX
play()
Connected
Speakers
BassBoostDecorator
BassBoostDecorator(BOOMBOX)
play()
TrebleBoostDecorator
TrebleBoostDecorator(BOOMBOX)
play()
1
1



Implementation without design pattern



Implementation with design pattern

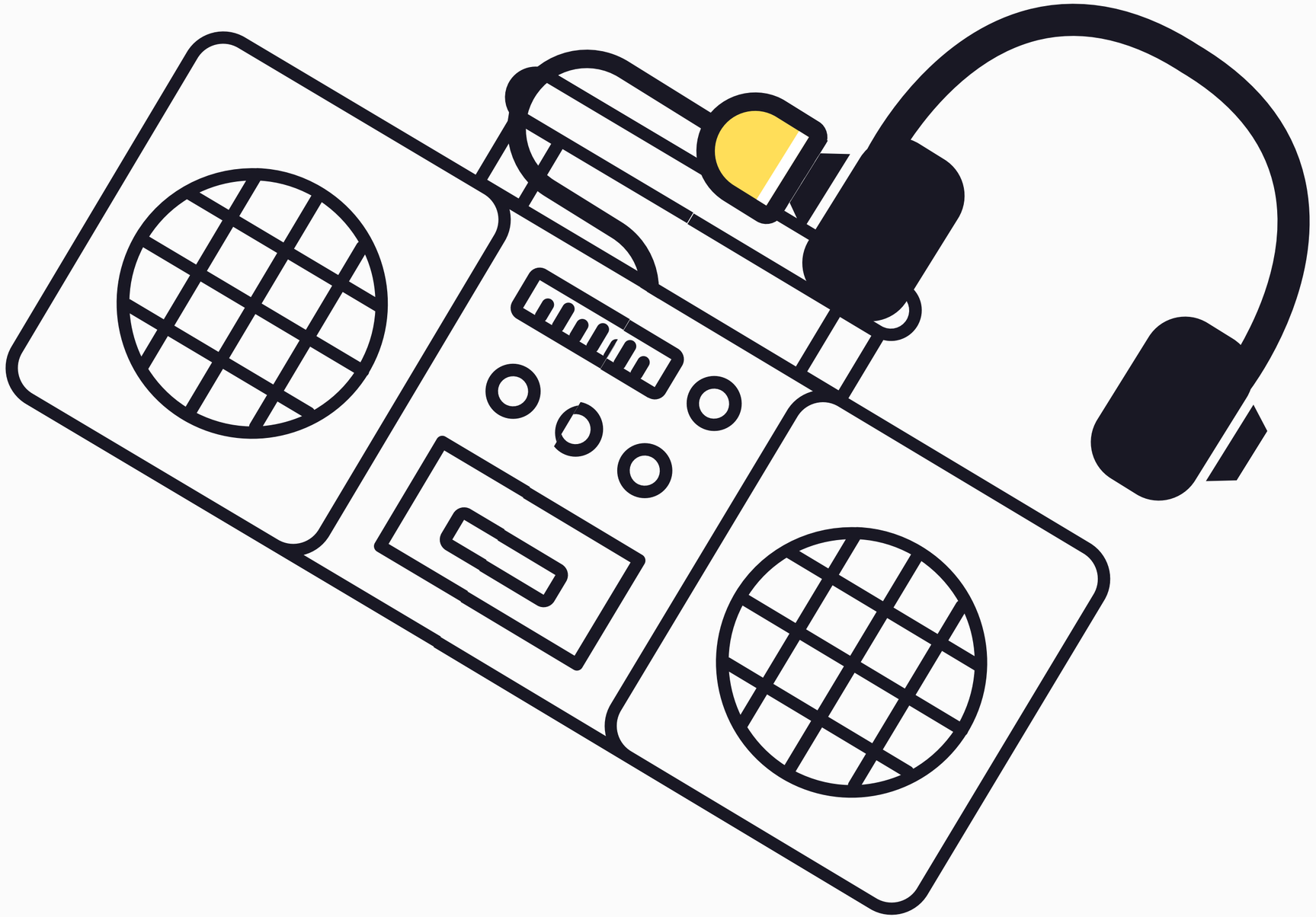


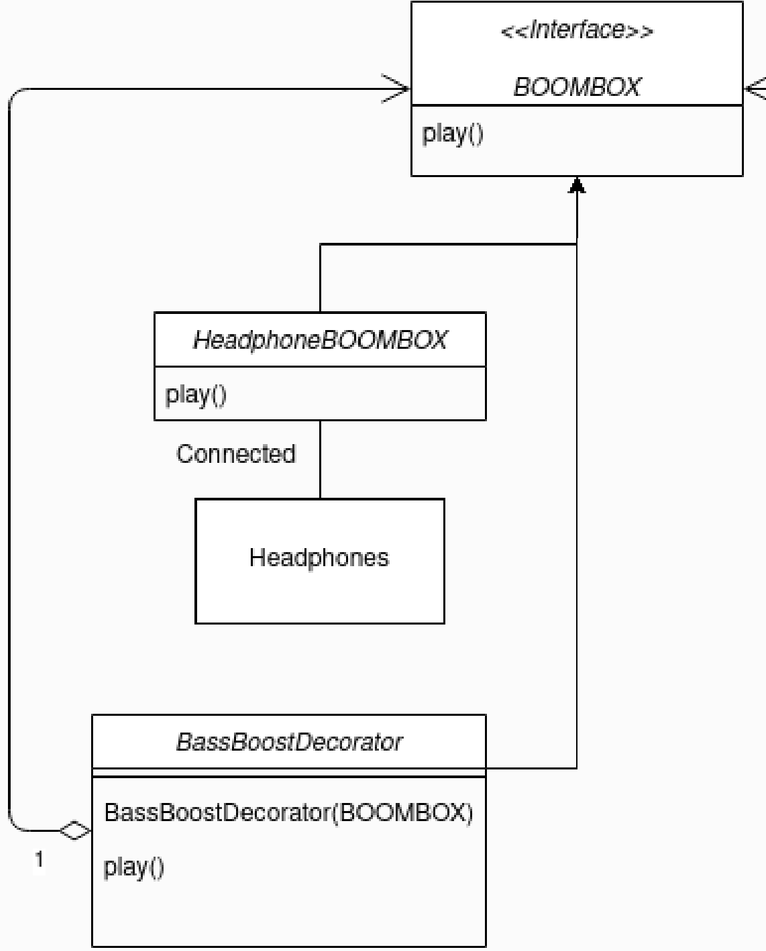
HeadphoneBOOMBOX

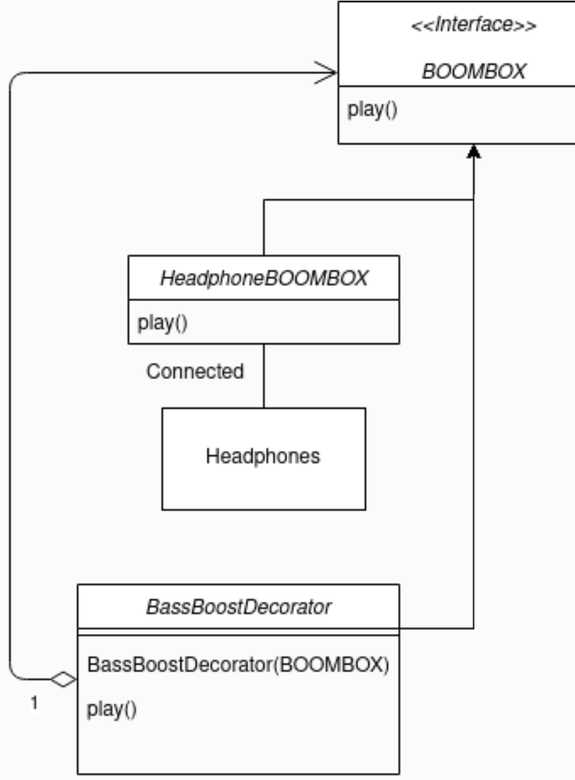
play()

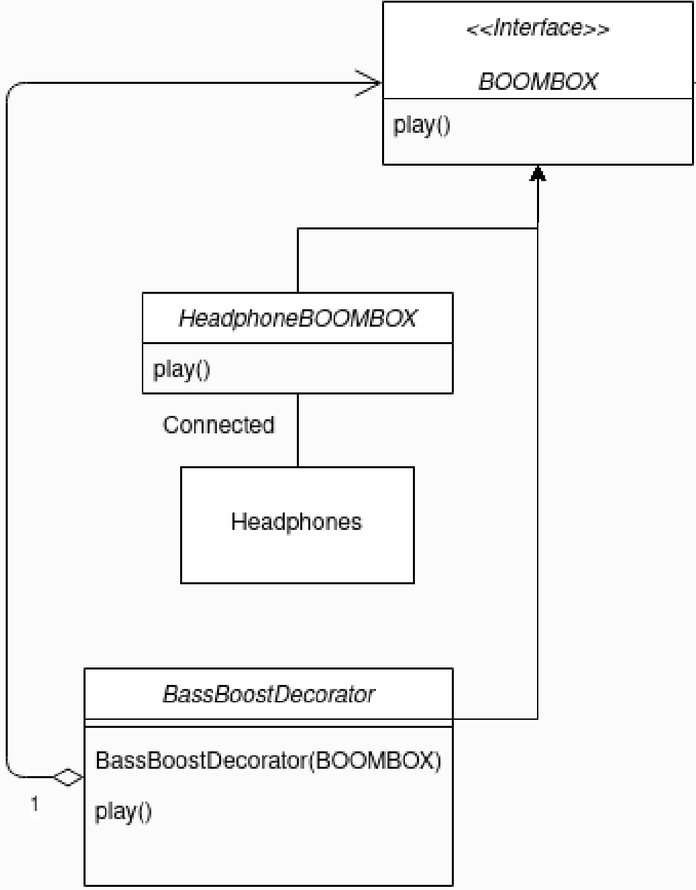
Connected

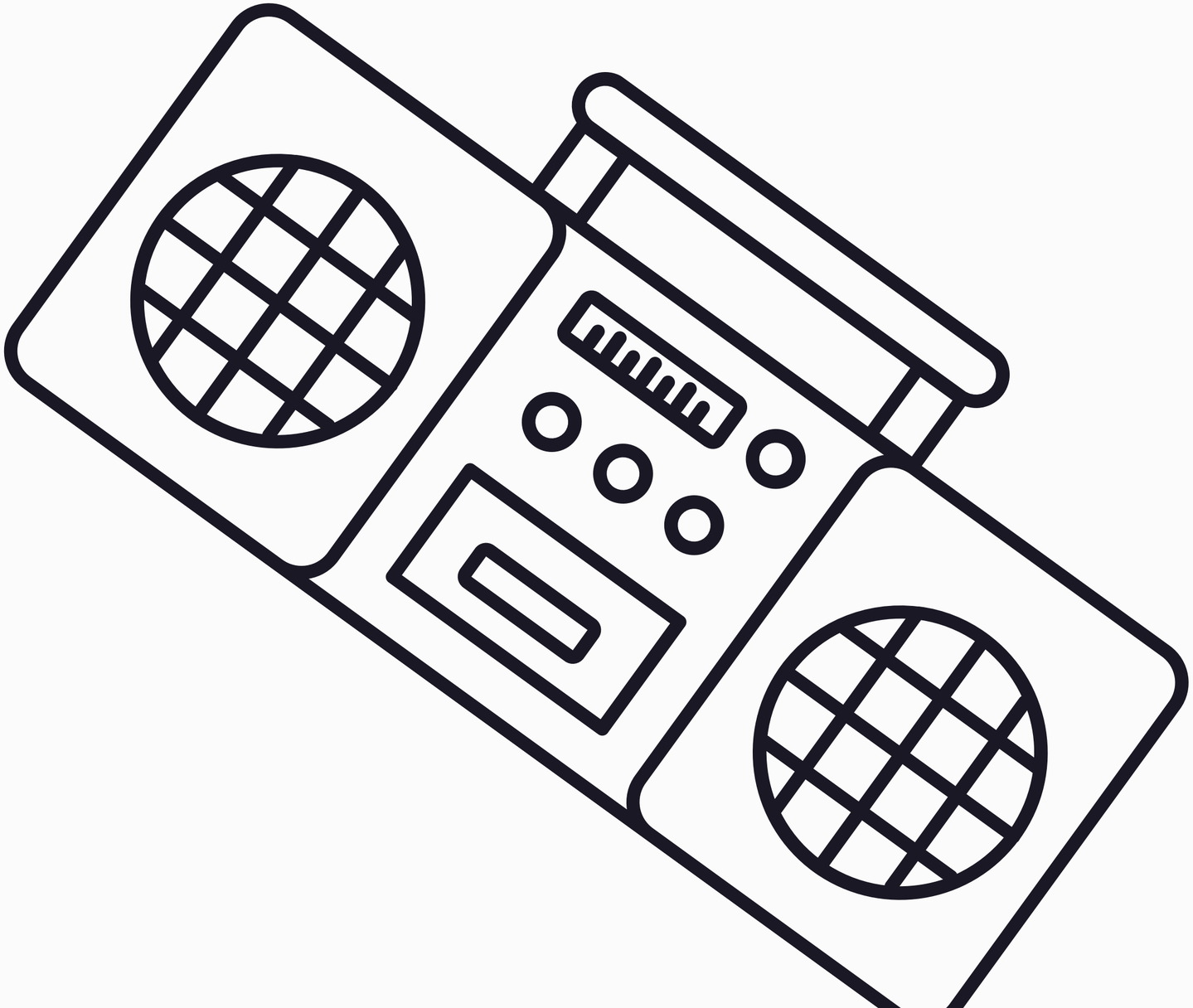
Headphones

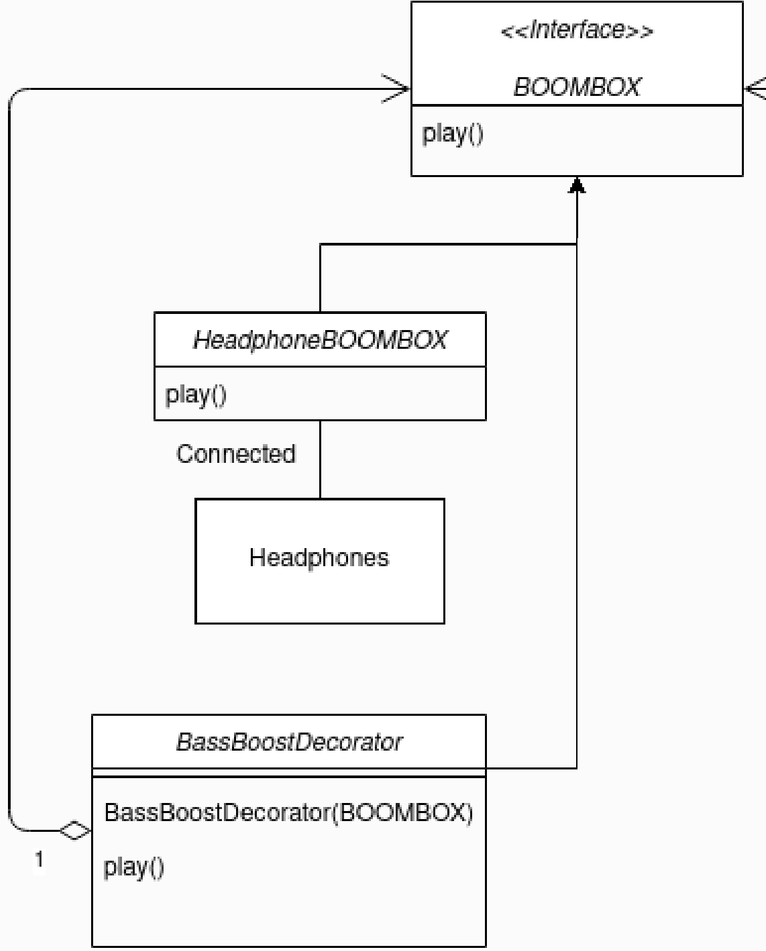



<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
BassBoostDecorator
BassBoostDecorator(BOOMBOX)
play()
1


<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
BassBoostDecorator
BassBoostDecorator(BOOMBOX)
play()
1


<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
BassBoostDecorator
BassBoostDecorator(BOOMBOX)
play()
1




<<Interface>>
BOOMBOX
play()
HeadphoneBOOMBOX
play()
Connected
Headphones
BassBoostDecorator
BassBoostDecorator(BOOMBOX)
play()
1

```
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 }
```

Implementation without design pattern

```java
public interface Boombox {
    SoundSignal music;
    void play();
}

public class HeadphoneBoombox implements Boombox{
    void play(){
        //I am playing this on headphones
        Headphones.play(this.music)
    }
}

public class BassBoostDecorator extends Boombox{
    Boombox boombox;
    public BassBoostDecorator(Boombox boombox){
        this.boombox = boombox;
    }
    void play(){
        boombox.music.bases++;
        boombox.play();
    }
}
```

Implementation with design pattern

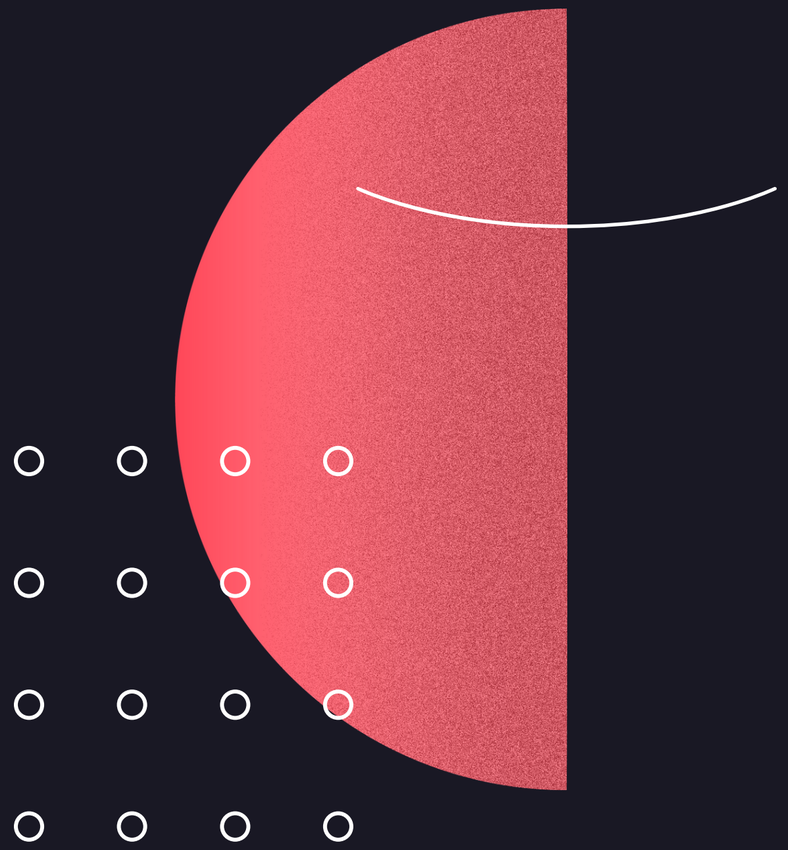
```java
public interface Boombox {
    SoundSignal music;
    void play();
}

public class HeadphoneBoombox implements Boombox{
    void play(){
        //I am playing this on headphones
        Headphones.play(this.music)
    }
}
```

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

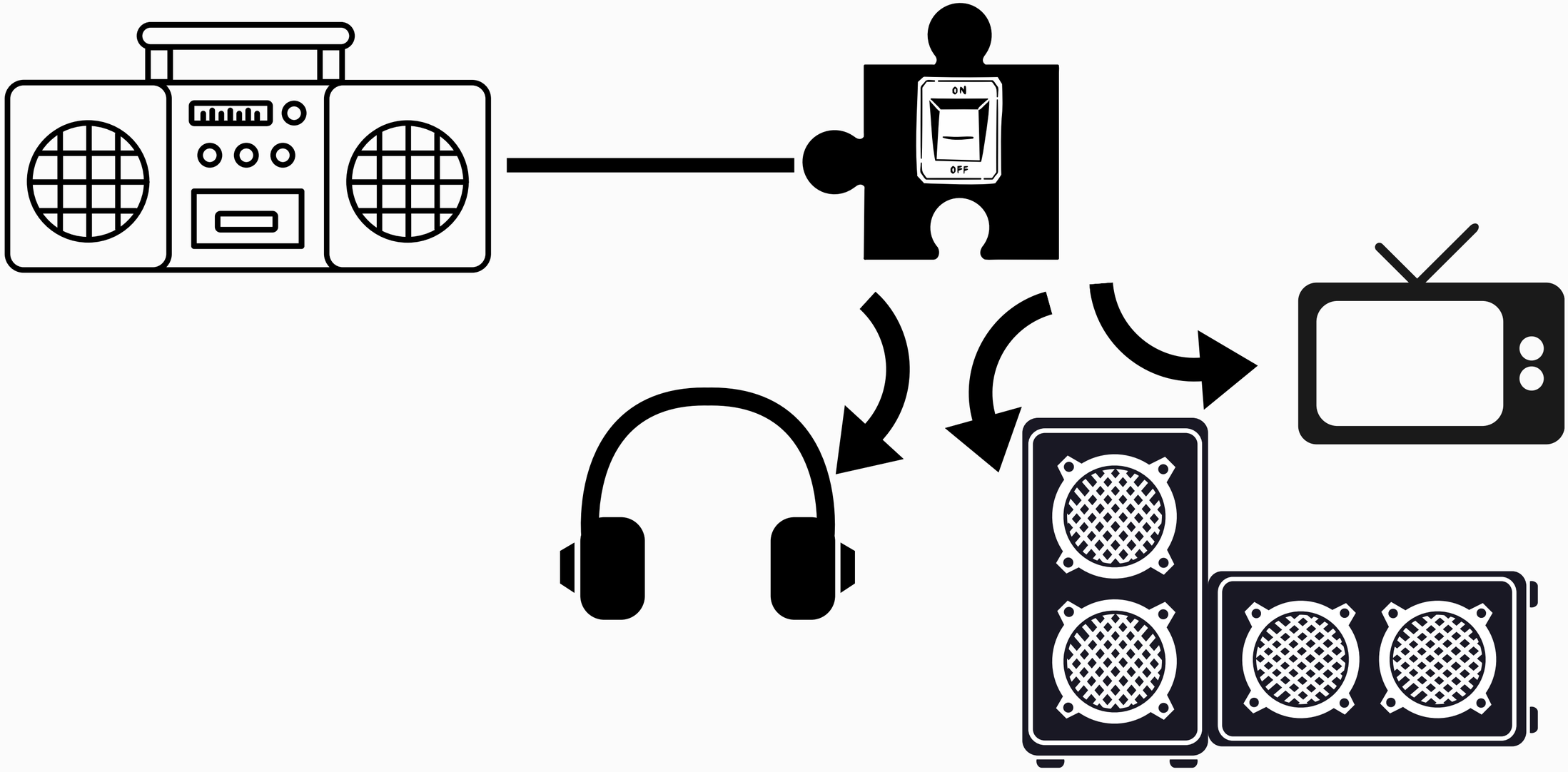
Implementation without design patterns

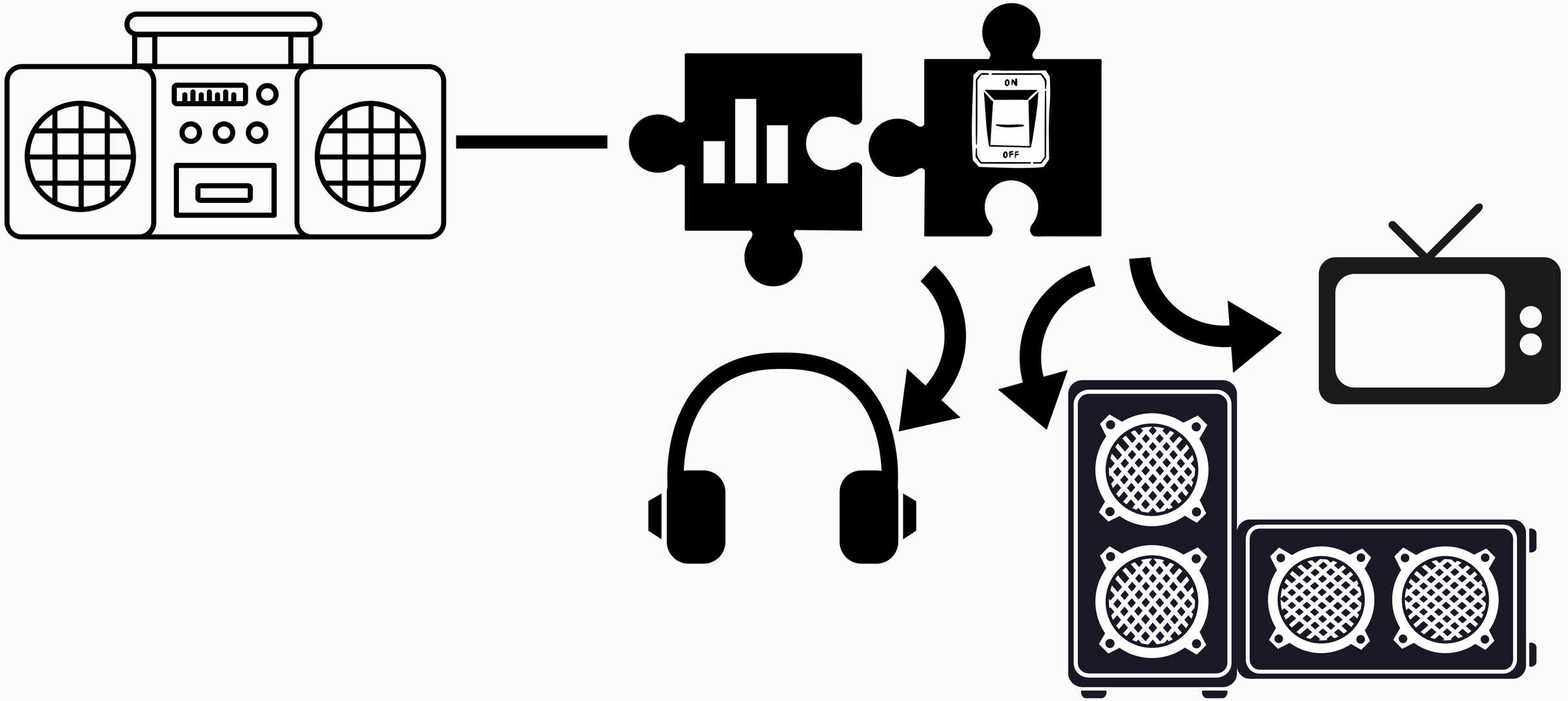
```java

public class BassBoostDecorator extends Boombox{
    Boombox boombox;
    public BassBoostDecorator(Boombox boombox){
        this.boombox = boombox;
    }
    void play(){
        boombox.music.bases++;
        boombox.play();
    }
}
```

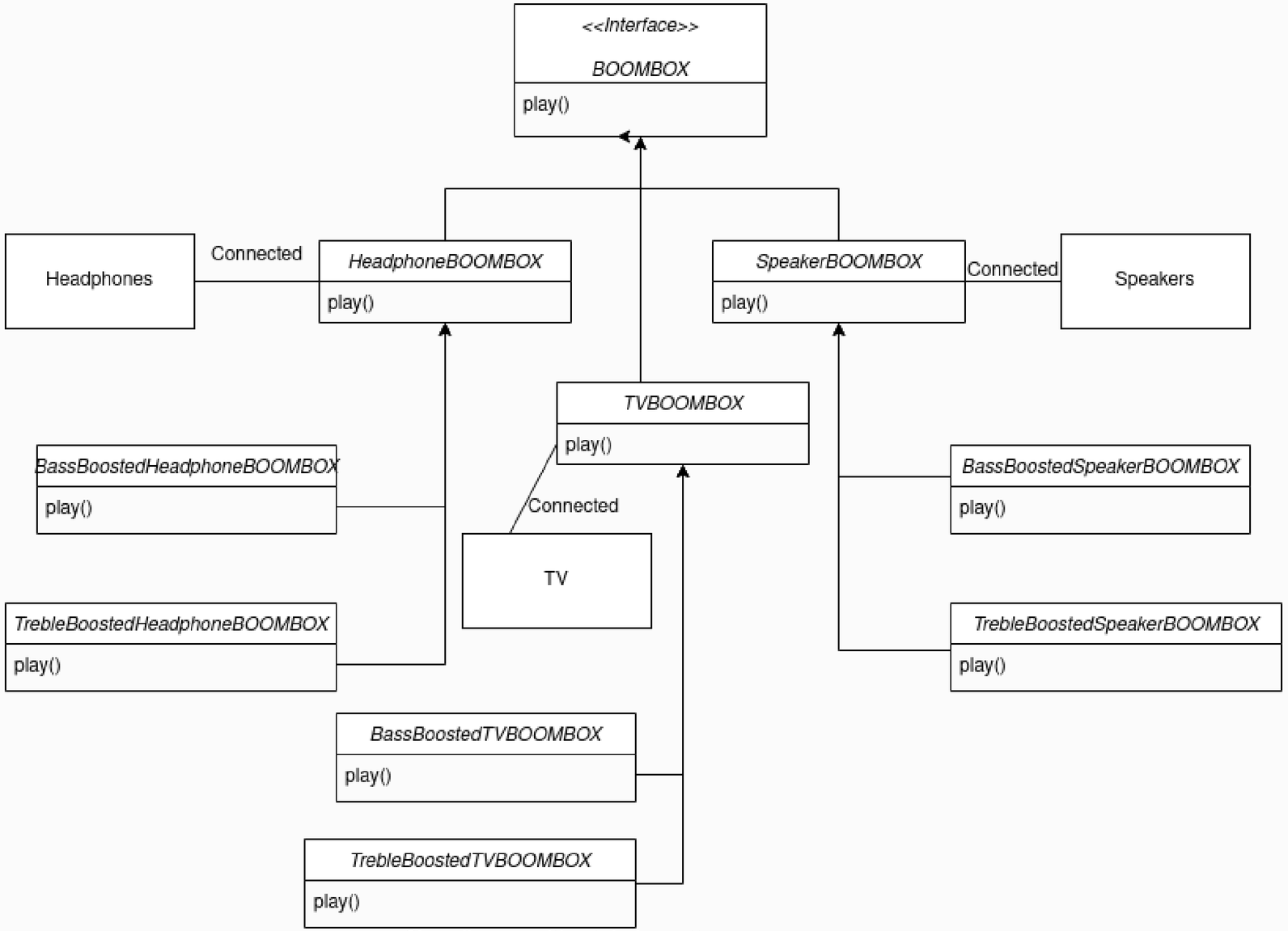
Implementation with design patterns



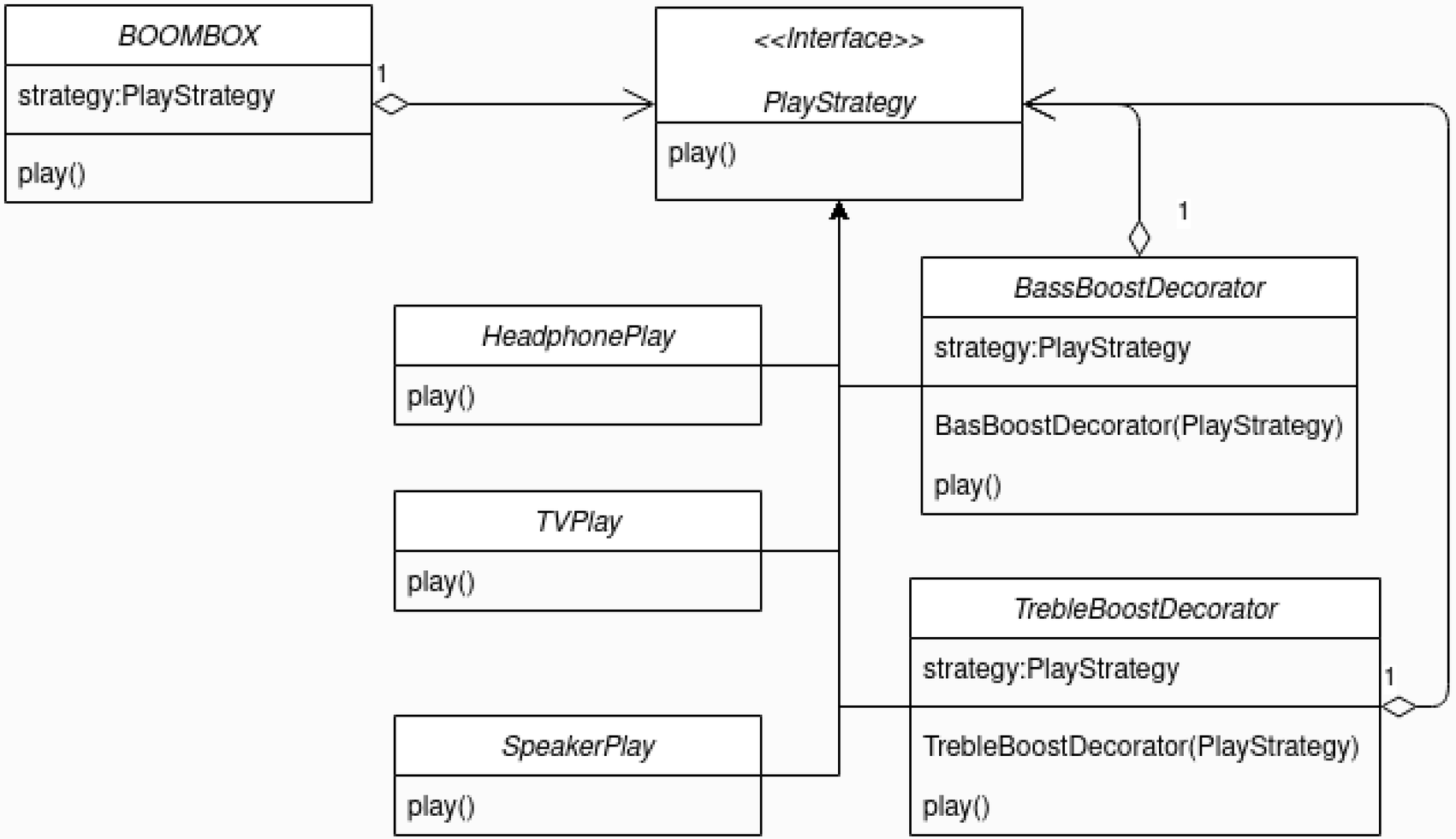
Strategy and Decorator Patterns are very powerful when used together


ON
OFF

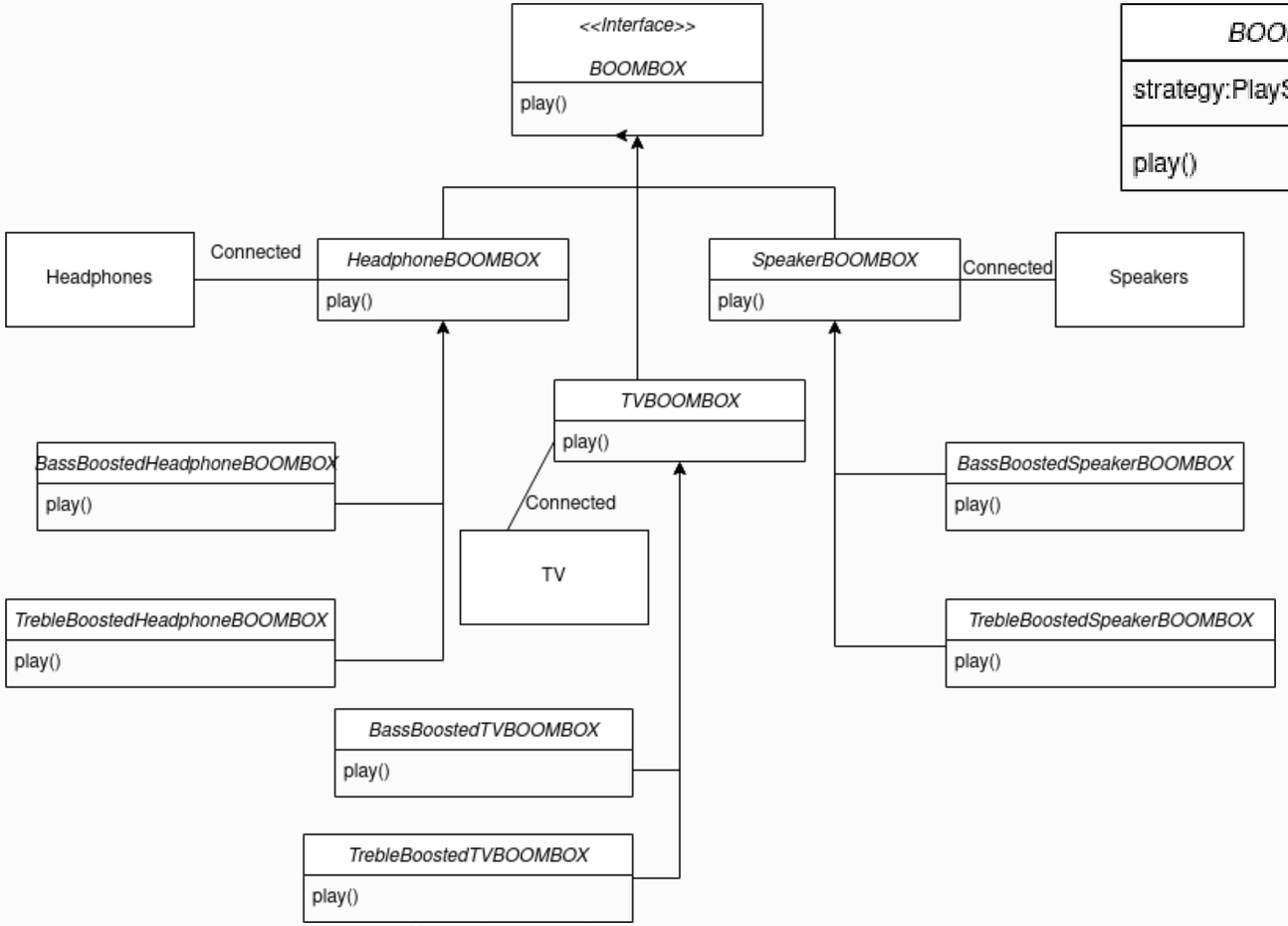

ON
OFF



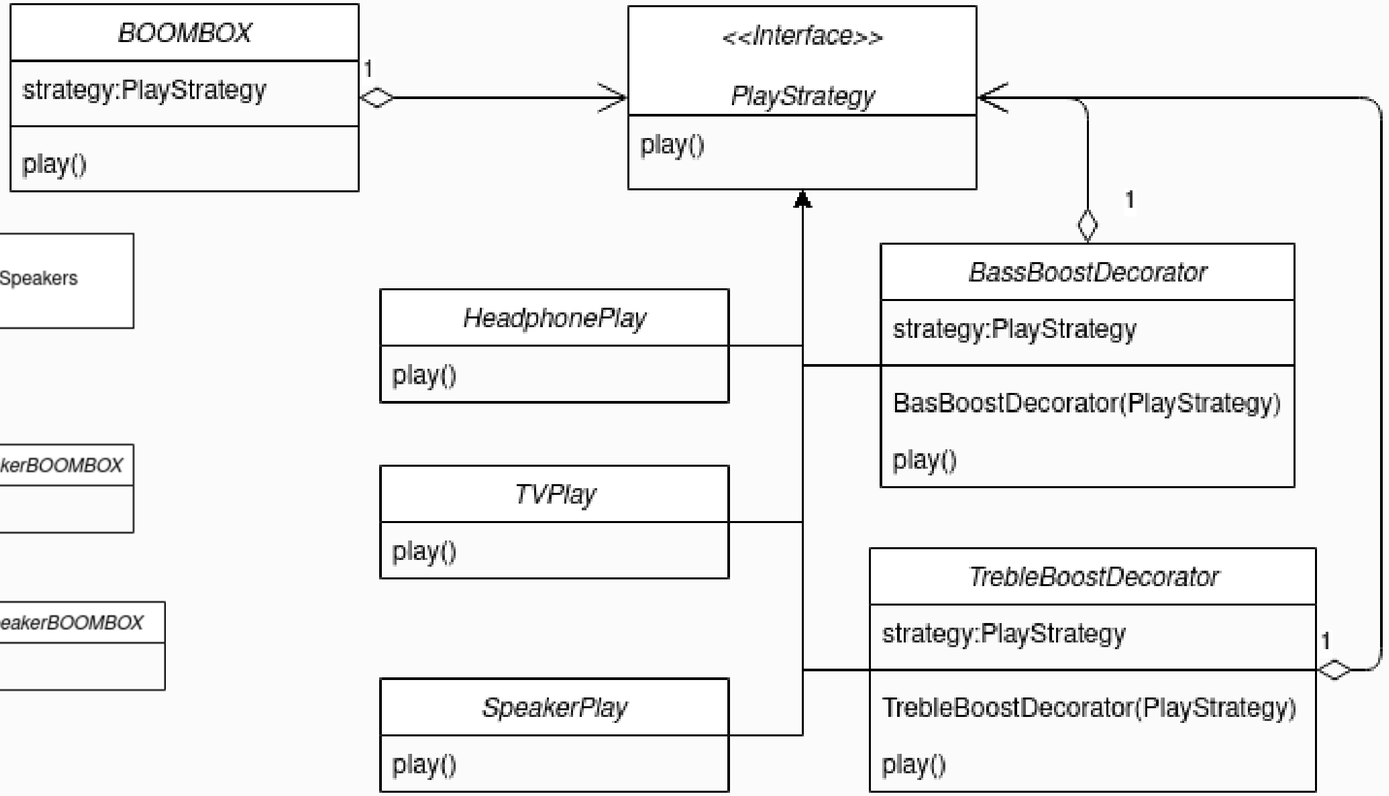
Class diagram without design patterns



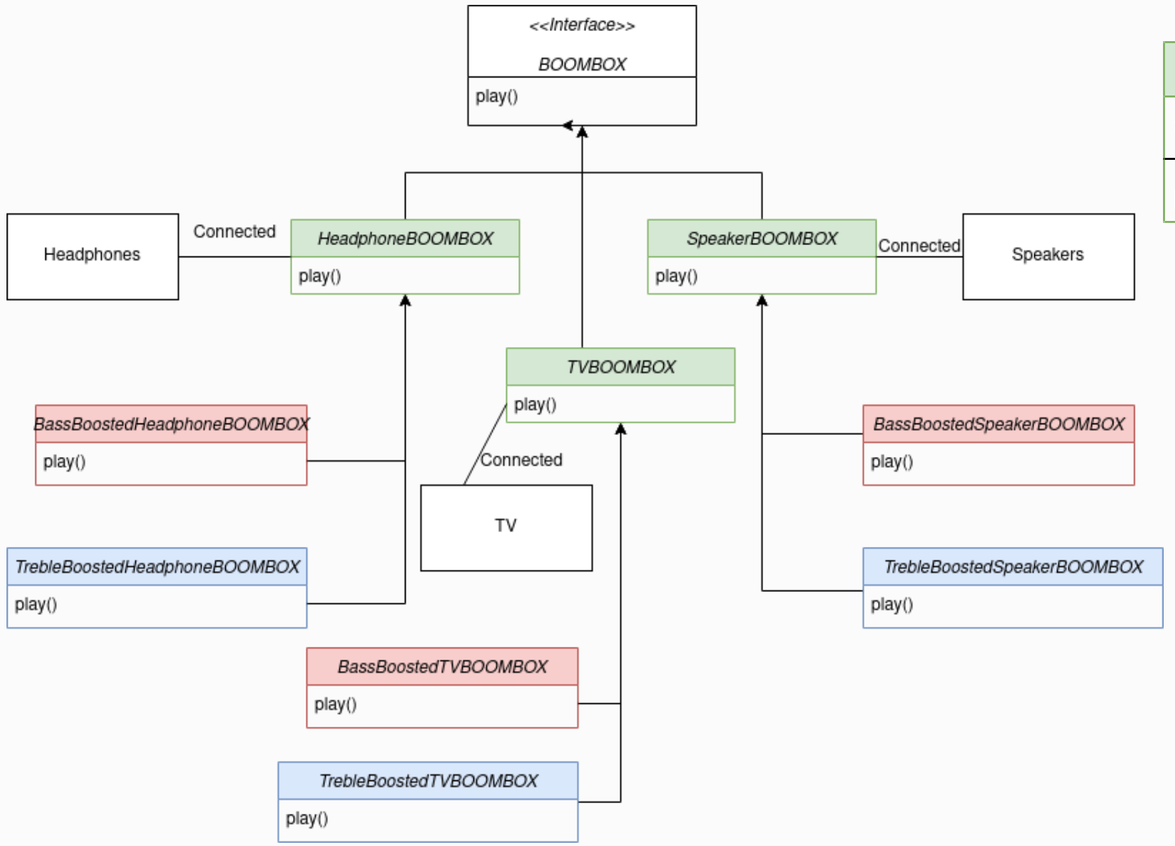
Class diagram with design patterns



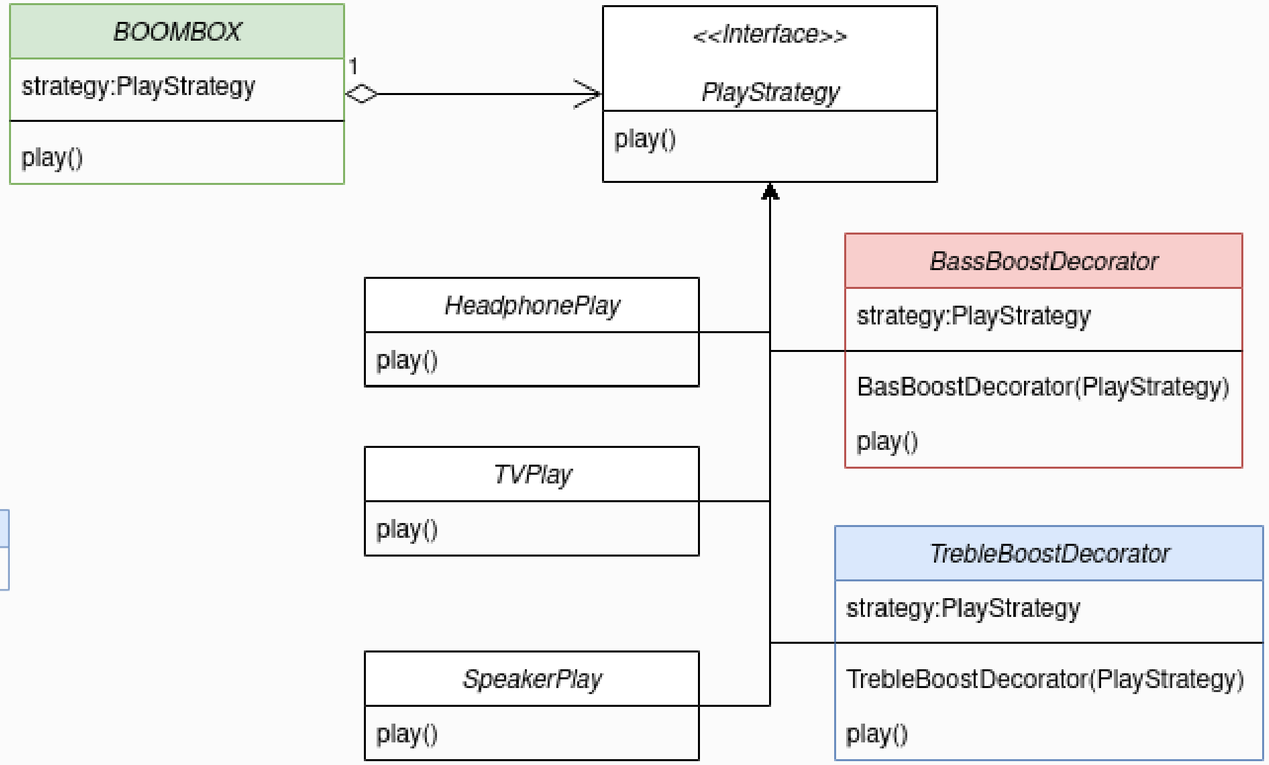
Class diagram without design patterns



Class diagram with design patterns



Class diagram without design patterns



Class diagram with design patterns


13

(0,0)

(width,height)

fill(r,g,b)

rect(x,y,width,height)

New assignment

Create two enemies,

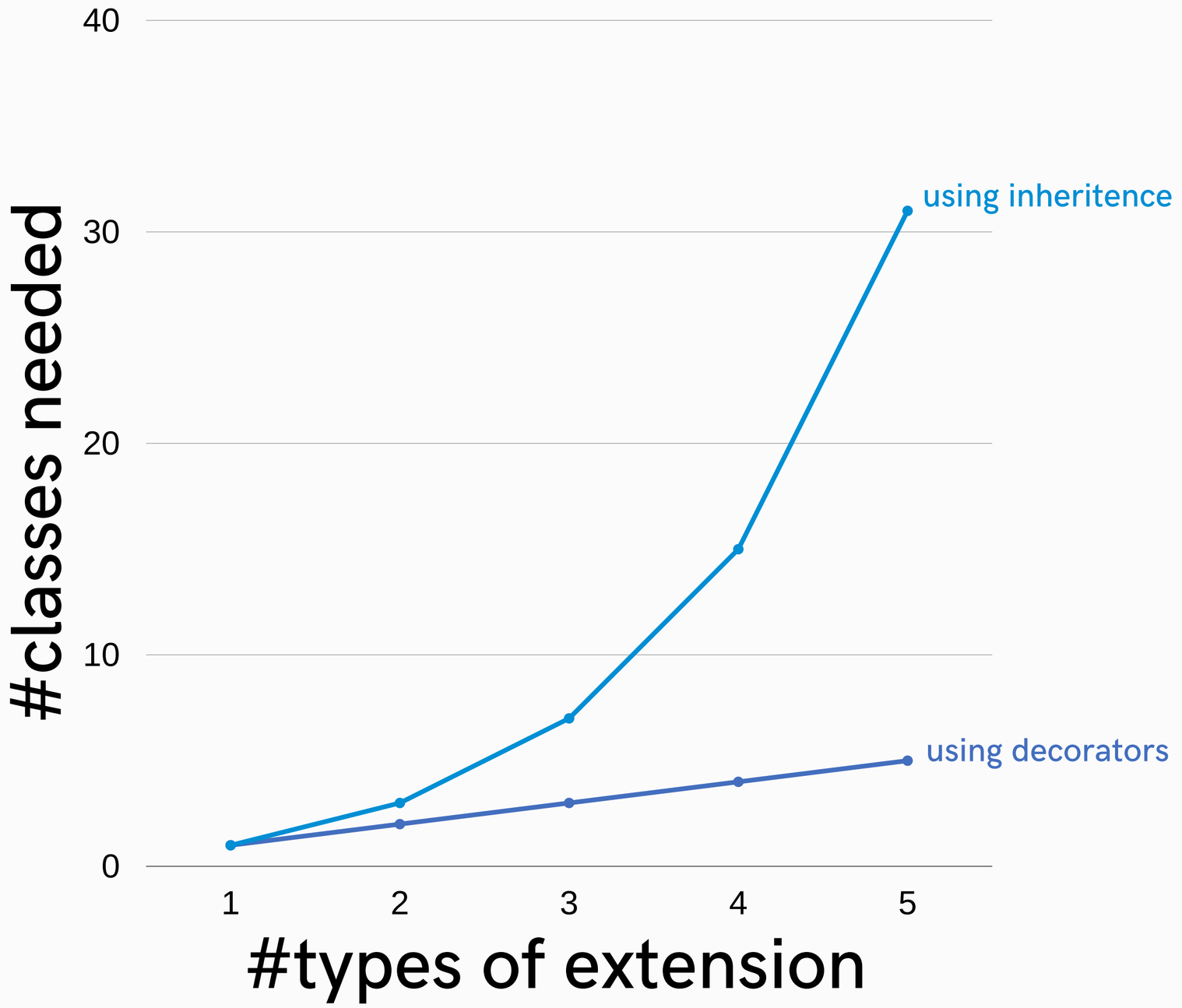
- Bouncing Enemy: Inverses the speed when hits a side
- Random Enemy: Randomizes the speed when hits a side

Both looking the same a blue square with side 50

New assignment

Create an extension of enemies

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


40
30
20
10
0
#classes needed
using inheritence
using decorators
1
2
3
4
5
#types of extension