ADAPTER & FACADE DESSIGN PATTERN

WELCOME TO OUR GROUP

Hao Trinh The

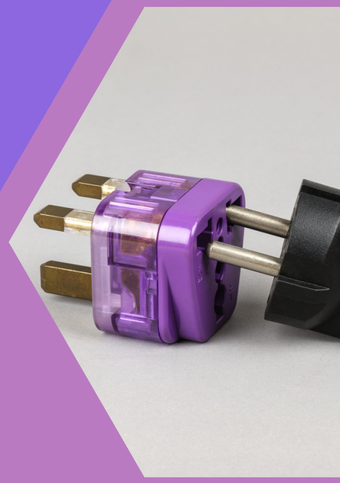
10421017

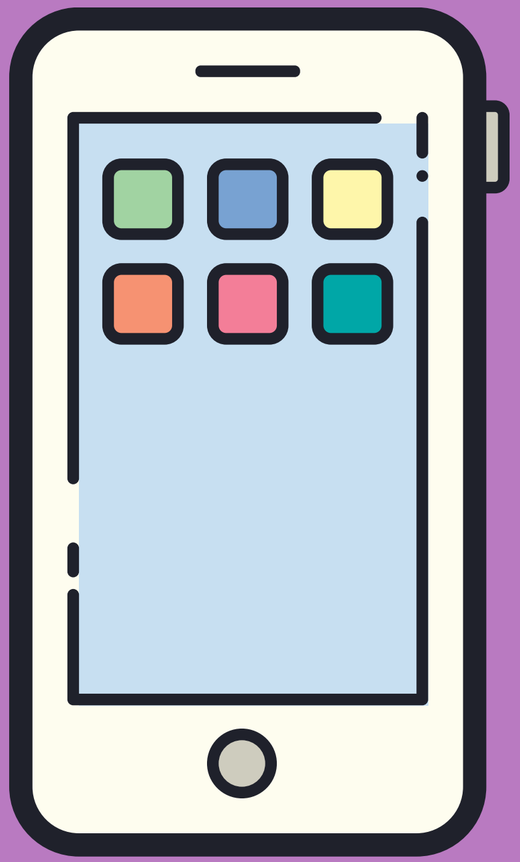
Huy Vo Vuong Bao

10421021

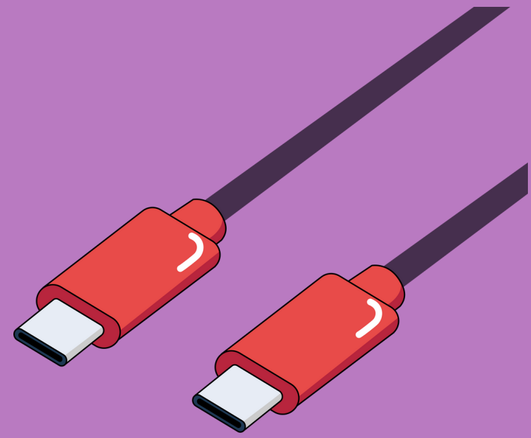
TABLE OF CONTENT

The Problem

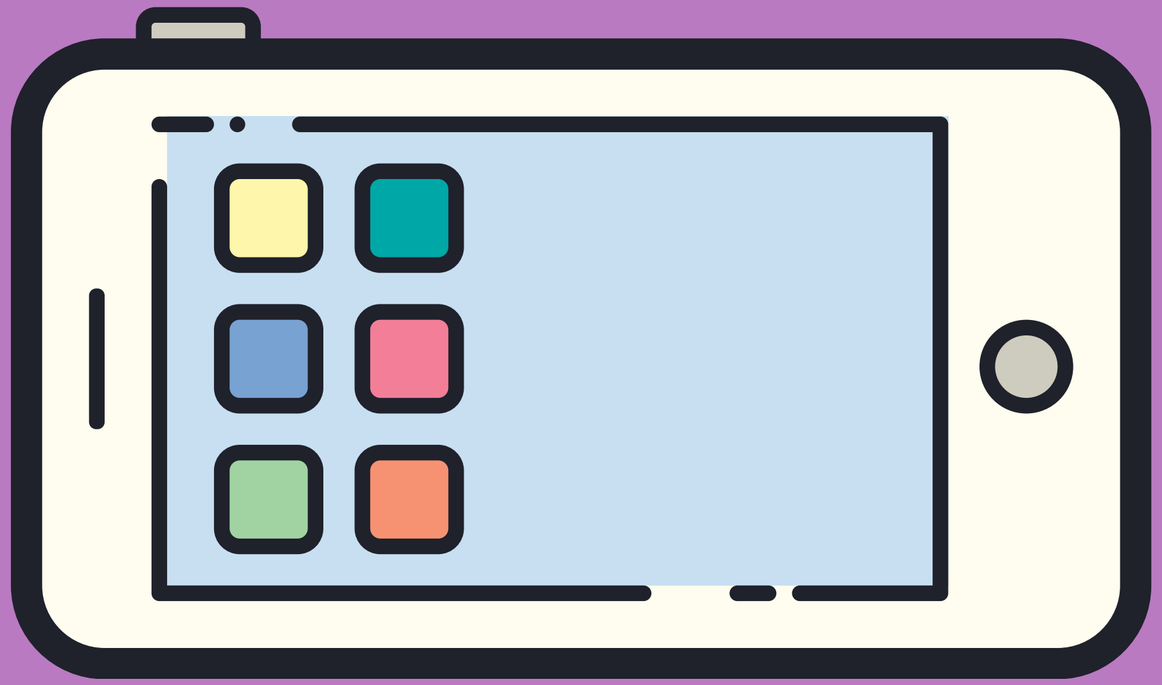




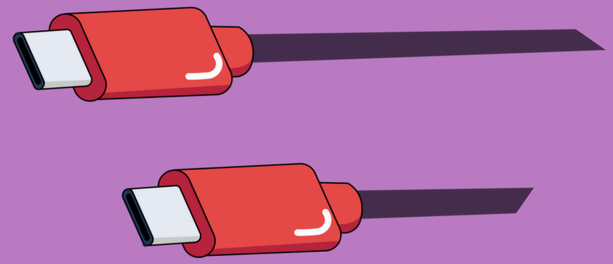
Iphone 5

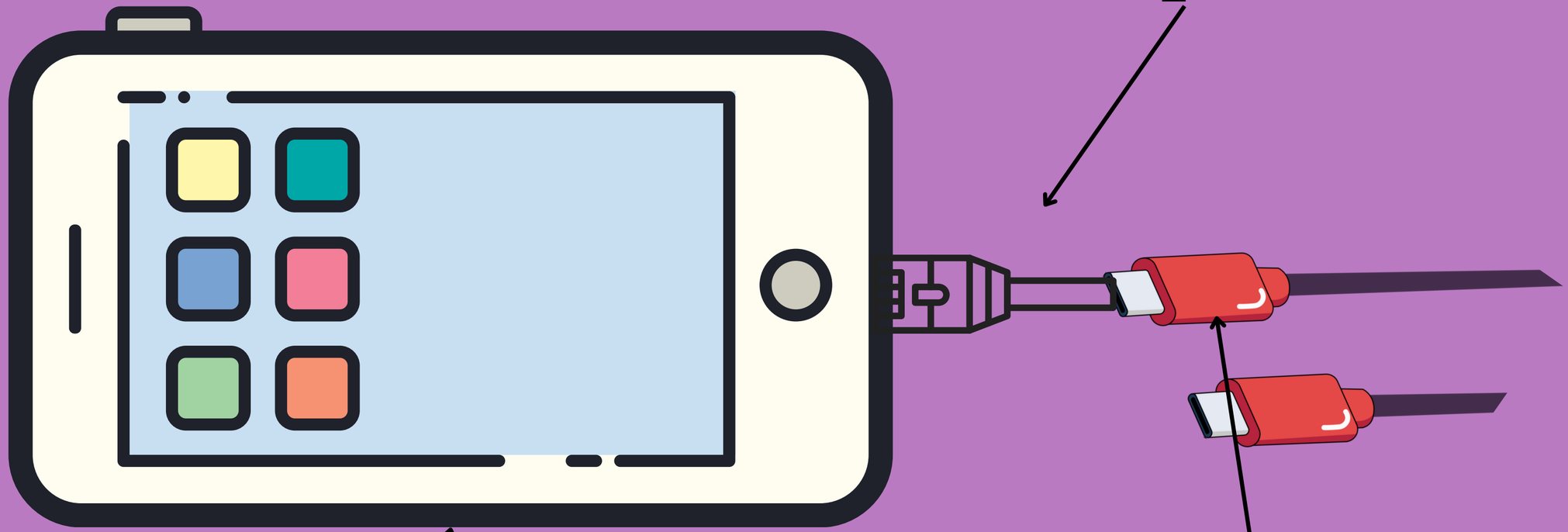


Type C cable



??

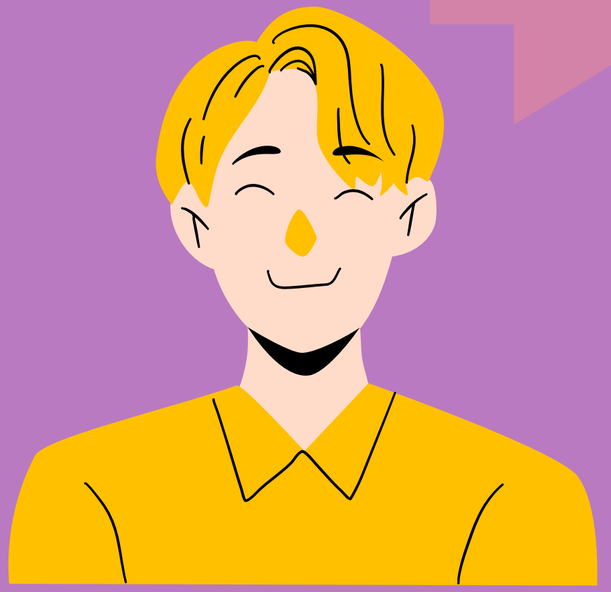



Adapter:))
Iphone 5
Type C
cable

DIFFERENT LANGUAGES

ENG

DE



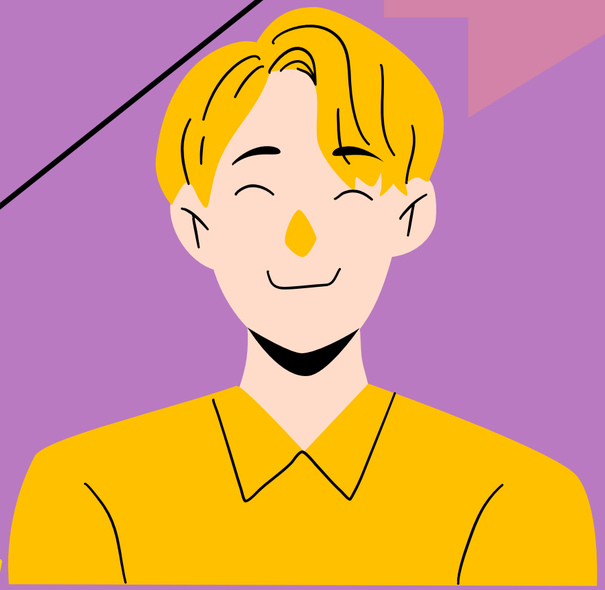
ENG

DE

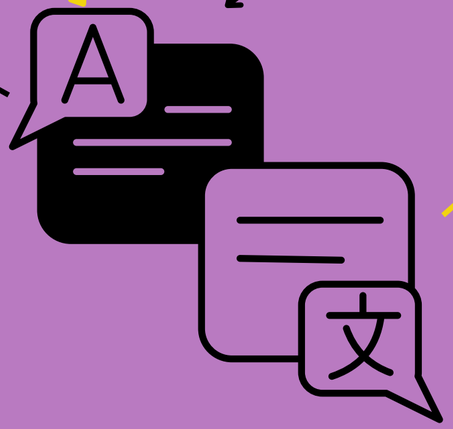
ENG

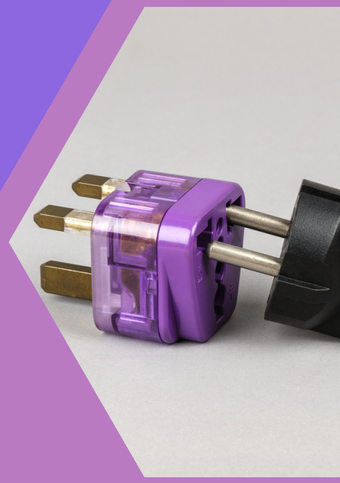
ENG

DEU



DEU





Adapter Definition

Adapter is a structural design pattern which allows imcompatible objects to collaborate

IMPLEMENTATION

```cpp
class Iphone {
private:
    string cableType = "lightning";
public:
    Iphone() = default;
    ~Iphone() = default;

    string getCableType() const {
        return cableType;
    }

    virtual string solution() const {
        return "Charging requires Lightning-type cable.";
    }
};

class CableTypeC {
public:
    CableTypeC() = default;
    ~CableTypeC() = default;

    string rightCableSolution(const string& cableTest){
        if(cableTest == "typeC")
            return "[!] Charging complete.";
        else
            return "[X] Charging failed. Wrong cable type.";
    }
};
```

```cpp
void Client(const Iphone* clientsPhone){
    cout << clientsPhone->solution() << endl;
}

int main(){
    Iphone* clientsPhone = new Iphone;
    Client(clientsPhone);
    cout<<endl;

    CableTypeC* cable = new CableTypeC;
    cout << cable->rightCableSolution(clientsPhone->getCableType()) << endl;
    cout<<endl;
```

```
Charging requires Lightning-type cable.

[X] Charging failed. Wrong cable type.
```

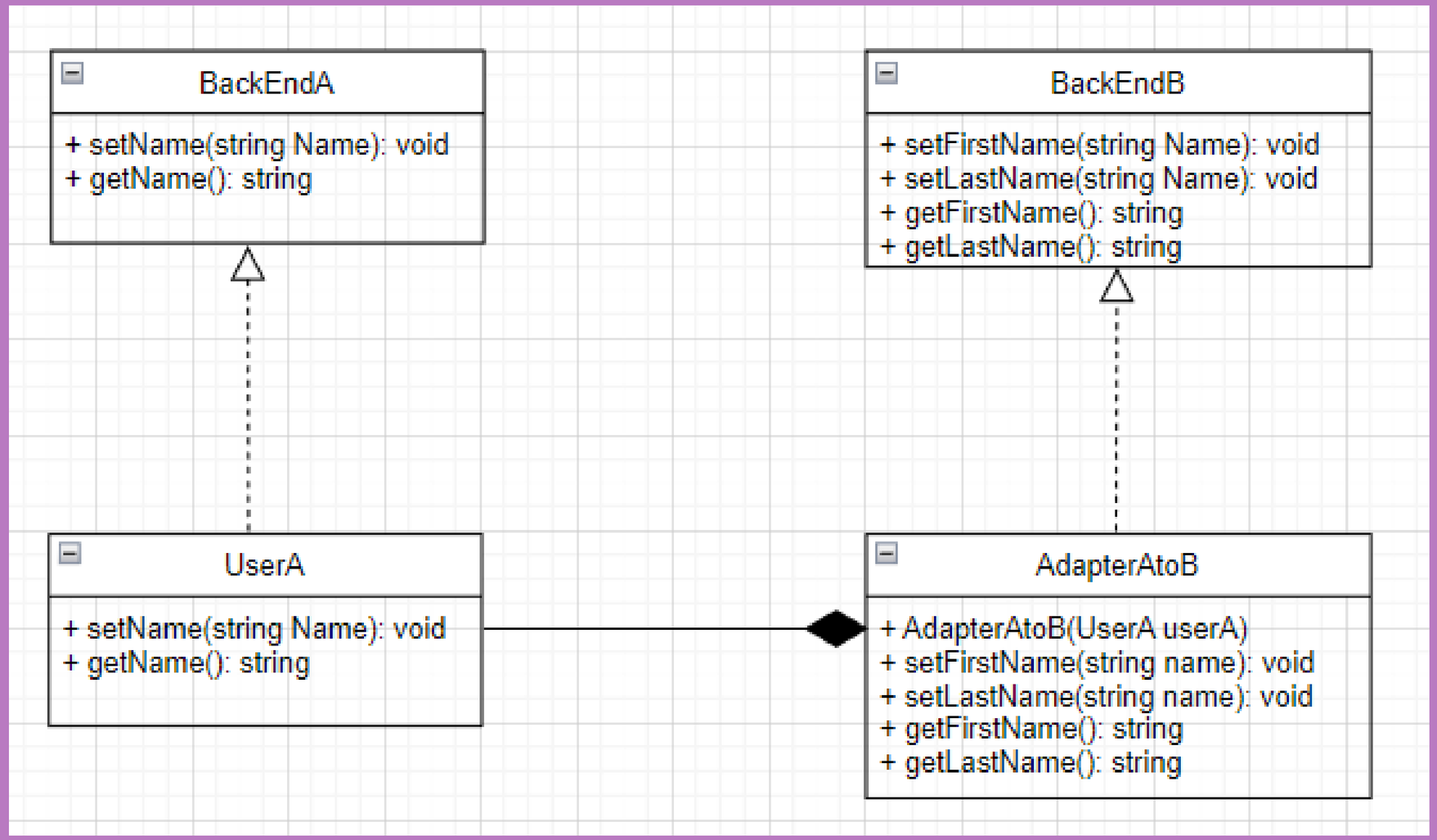
IMPLEMENTATION

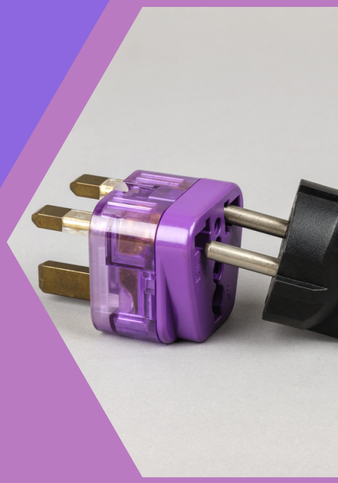
```
class Adapter : public Iphone {
private:
    CableTypeC* cable;
public:
    Adapter(CableTypeC* cable) : cable(cable) {}
    ~Adapter() = default;

    string solution() const {
        if(getCableType() != "typeC")
            return cable->rightCableSolution("typeC");
        else
            return cable->rightCableSolution(getCableType());
    }
};
```

```
Adapter* adapter = new Adapter(new CableTypeC);
Client(adapter);
```

```
[!] Charging complete.
```

MORE IMPLEMENTATION





Summary

The adapter pattern convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of the incompatible interfaces.

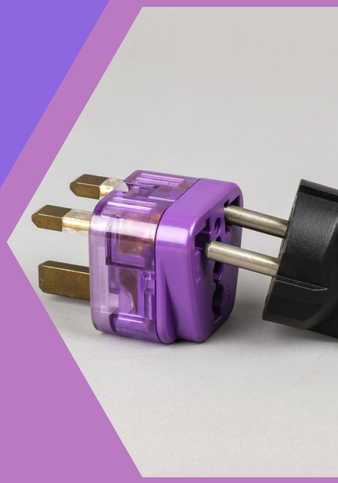
REMARK!!!

Advantages:

- Helps achieve reusability and flexibility.
- Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

Disadvantages:

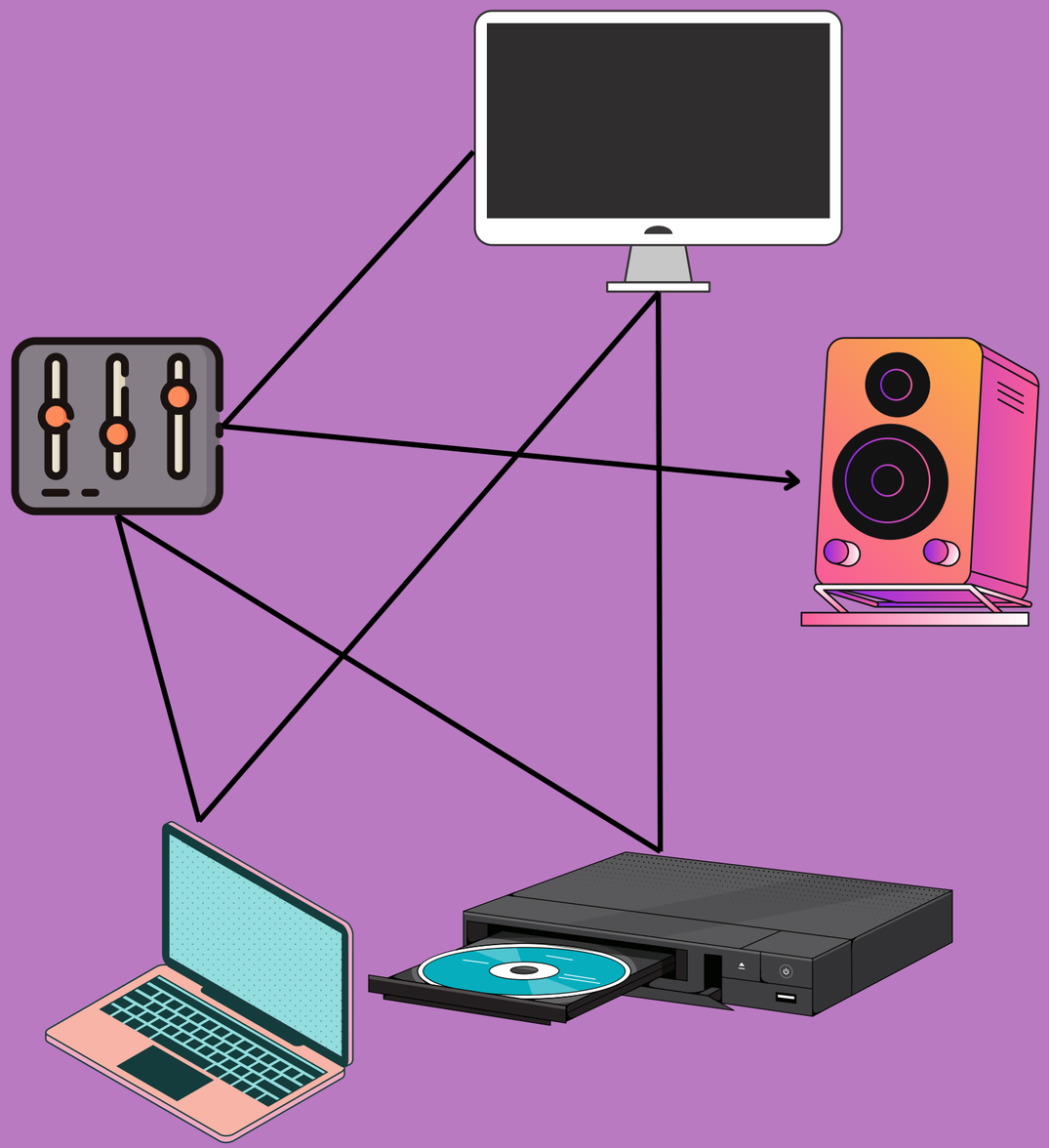
- All requests are forwarded, so there is a slight increase in the overload.
- Sometimes many adaptations are requied along an adapter chain to reach the type which is required



Facade

A Structural Pattern

REAL LIFE PROBLEM



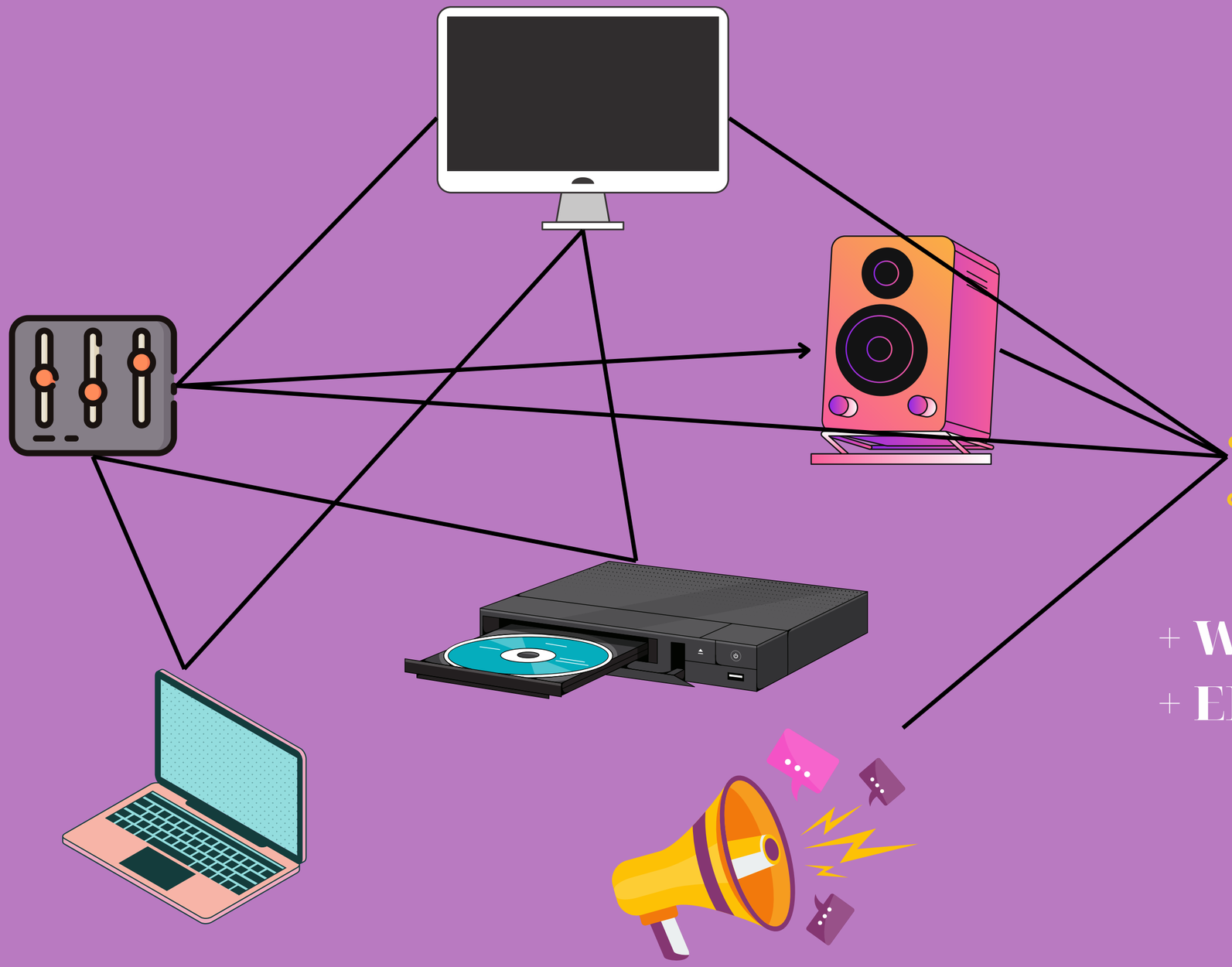
Steps to watch a movie

1) Turn the DVD player on
2) Turn the Screen on
3) Turn the Amplifier on
4) Turn the speaker on
5) Set the amplifier to DVD input
6) Set the Amplifier volume to medium
7) Switching the light off

REAL LIFE PROBLEM

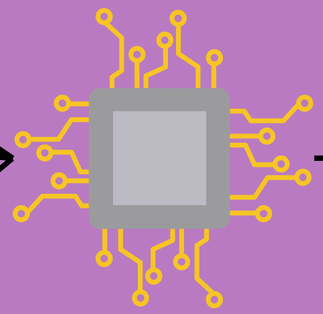
STEP TO END A MOVIE:))

REAL LIFE PROBLEM



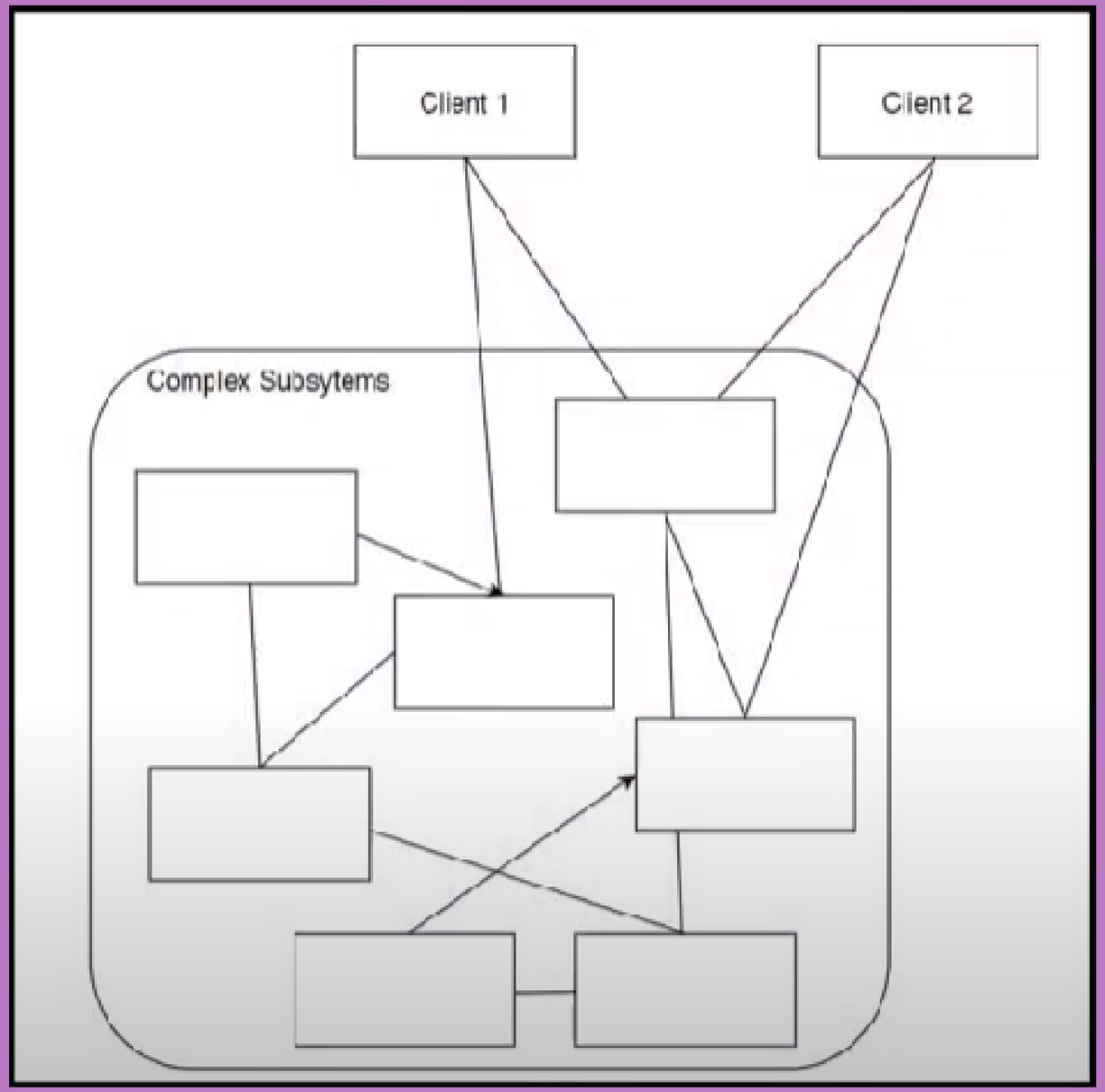
Theater
Facade

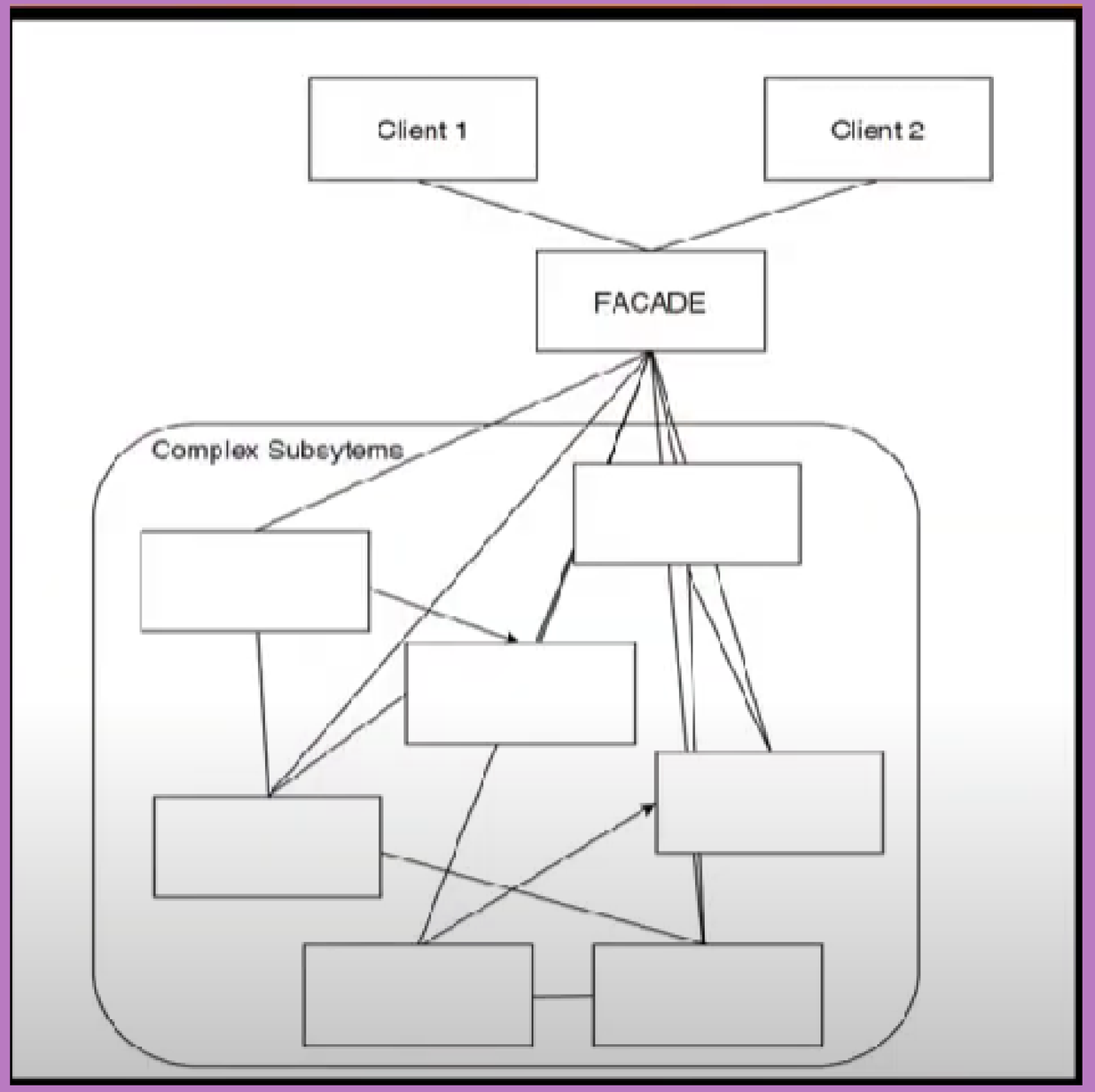
Client

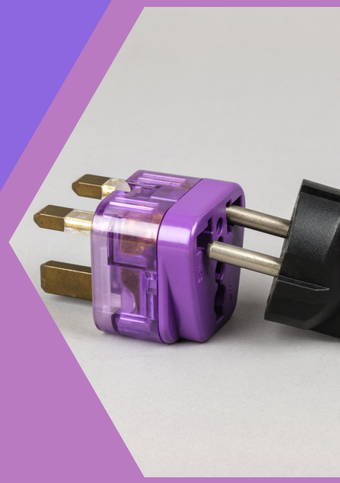


+ WATCH_MOVIE()
+ END_MOVIE()

IN PROGRAMMING







Facade Definition

Facade is structural design pattern that provides a simplified(but limited) interface to a complex system of classes, library or framework.

IMPLEMENTATION

```
class DVDPlayer {
    public:
        string turnOn() const { return "DVD Player is turned on!\n"; }
        string turnOff() const { return "DVD Player is turned off!\n"; }
        string pause() const { return "DVD Player is paused!\n"; }
};
```

```
class Screen {
    public:
        string turnOn() const { return "Screen is turned on!\n"; }
        string turnOff() const { return "Screen is turned off!\n"; }
};
```

```
class Amplifier {
    public:
        string turnOn() const { return "Amplifier is turned on!\n"; }
        string turnOff() const { return "Amplifier is turned off!\n"; }
        string setDVDinput() const { return "Amplifier DVD Input Mode!\n"; }
        string setCDinput() const { return "Amplifier CD Input Mode!\n"; }
        string volumeLow() const { return "Amplifier Volume is Low!\n"; }
        string volumeMedium() const { return "Amplifier Volume is Medium!\n"; }
        string volumeHigh() const { return "Amplifier Volume is High!\n"; }

};
```

```
class Speaker {
    public:
        string turnOn() const { return "Speaker is turned on!\n"; }
        string turnOff() const { return "Speaker is turned off!\n"; }
};
```

```
class Light {
    public:
        string turnOn() const { return "Light is turned on!\n"; }
        string turnOff() const { return "Light is turned off!\n"; }
};
```

IMPLEMENTATION

```
class Facade {
    protected:
        DVDPlayer *dvdplayer_;
        Screen *screen_;
        Amplifier *amplifier_;
        Speaker *speaker_;
        Light *light_;

    public:
        Facade(
                DVDPlayer *dvdplayer = nullptr,
                Screen *screen = nullptr,
                Amplifier *amplifier = nullptr,
                Speaker *speaker = nullptr,
                Light *light = nullptr) {
            this->dvdplayer_ = dvdplayer ?: new DVDPlayer;
            this->screen_ = screen ?: new Screen;
            this->amplifier_ = amplifier ?: new Amplifier;
            this->speaker_ = speaker ?: new Speaker;
            this->light_ = light ?: new Light;
        }

        ~Facade() {
            delete dvdplayer_;
            delete screen_;
            delete amplifier_;
            delete speaker_;
            delete light_;
        }
```

IMPLEMENTATION

```
    string watchMovie() {
        string result = "Facade initializes Watching Movie:\n";
        result += this->dvdplayer_->turnOn();
        result += this->screen_->turnOn();
        result += this->amplifier_->turnOn();
        result += this->speaker_->turnOn();
        result += this->amplifier_->setDVDinput();
        result += amplifier_->volumeMedium();
        result += this->light_->turnOff();
        return result;
    }

};
```

IMPLEMENTATION

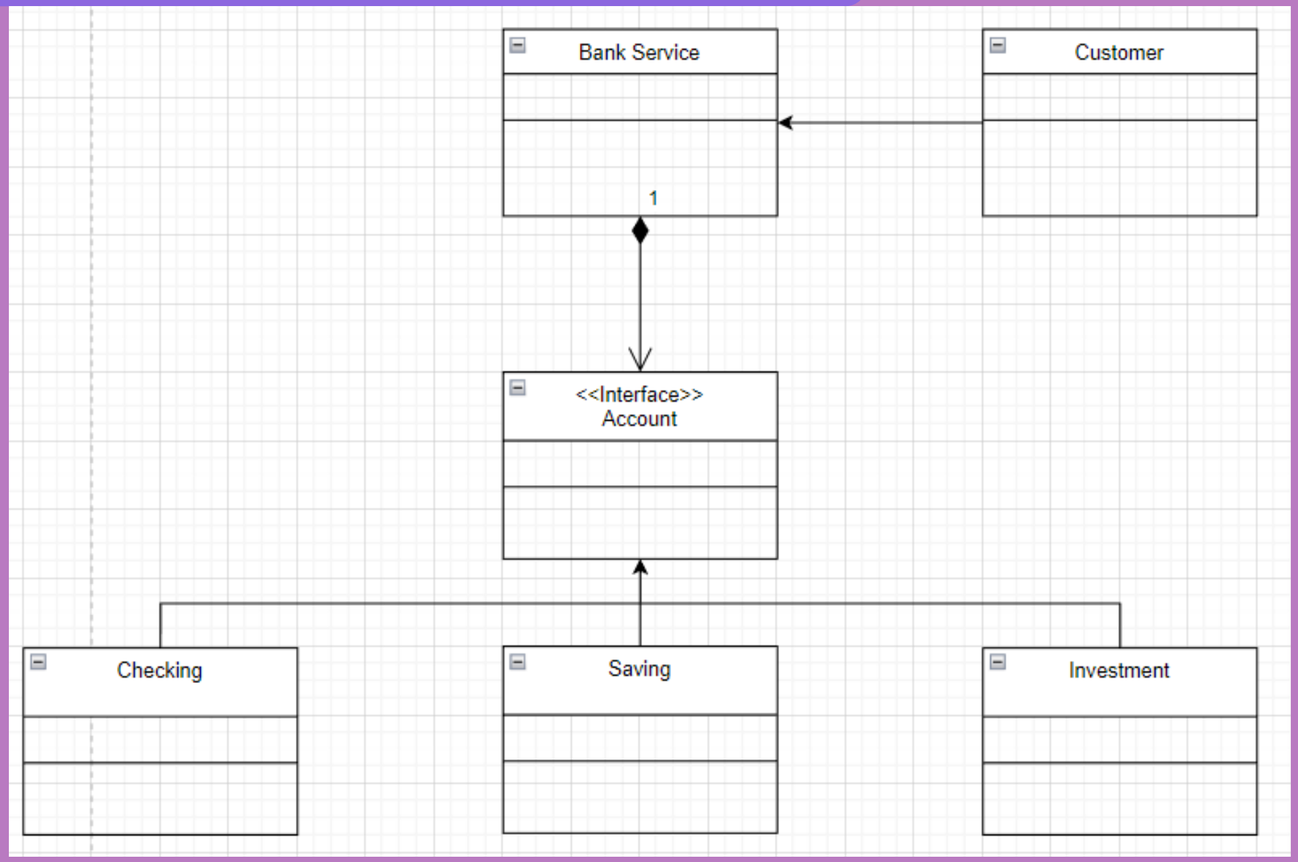
```
    string watchMovie() {
        string result = "Facade initializes Watching Movie:\n";
        result += this->dvdplayer_->turnOn();
        result += this->screen_->turnOn();
        result += this->amplifier_->turnOn();
        result += this->speaker_->turnOn();
        result += this->amplifier_->setDVDinput();
        result += amplifier_->volumeMedium();
        result += this->light_->turnOff();
        return result;
    }

};
```

IMPLEMENTATION

```
void Client(Facade *facade) {
    cout << facade->watchMovie();
}


int main() {
    DVDPlayer *dvdplayer = new DVDPlayer;
    Screen *screen = new Screen;
    Amplifier *amplifier = new Amplifier;
    Speaker *speaker = new Speaker;
    Light *light = new Light;

    Facade *facade = new Facade(dvdplayer, screen, amplifier, speaker, light);
    Client(facade);

    delete facade;
}
```

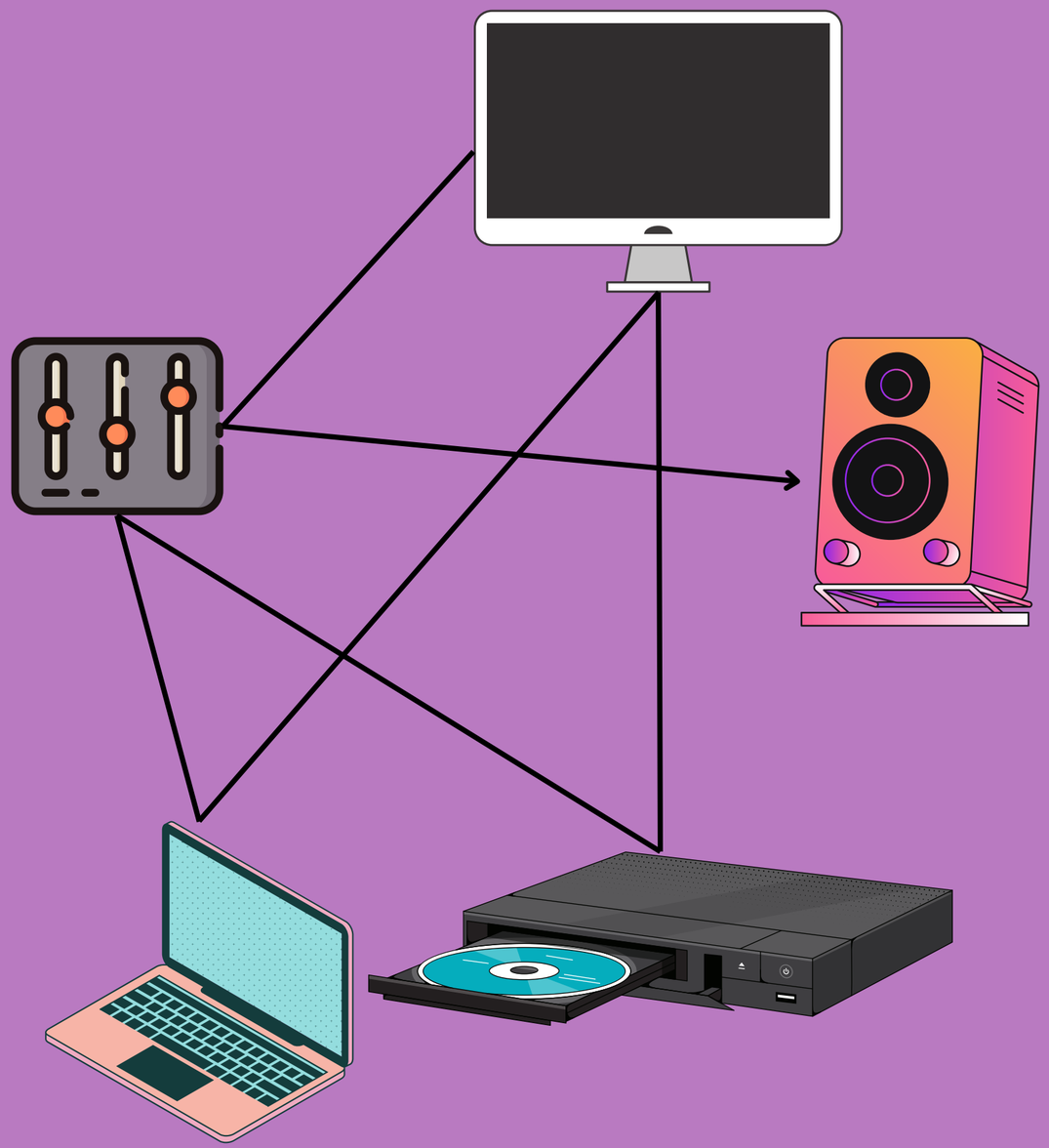
```
Facade initializes Watching Movie:
DVD Player is turned on!
Screen is turned on!
Amplifier is turned on!
Speaker is turned on!
Amplifier DVD Input Mode!
Amplifier Volume is Medium!
Light is turned off!
```

FACADE PATTERN APPLICATION



- We introduce the BankService clas to act as a Facade for the checking, saving and investment classes.
- The customer class no longer needs to handle instantiation or deal with any of the complexities of financial management.
- Since three different accounts, all implement the Account interface, BankService class is effectively wrapping the account interfacing classes, and presenting a simpler front to them for the customer client class to use.

REAL LIFE PROBLEM

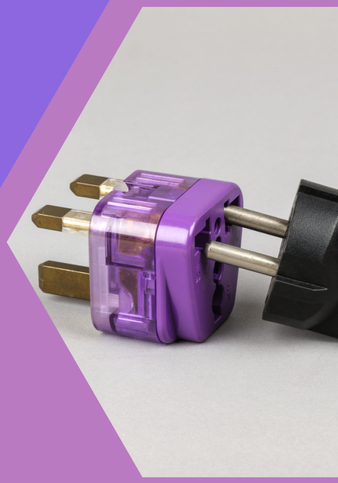


Steps to end a movie

1) Switching the light on
2) Turn the speaker off
3) Turn the amplifier off
4) Turn the screen off
5) Turn the DVD player off

Implement Facade Pattern to solve the above problem

Facade Summary



Pros:

1) Minimizes complexity of sub-systems
2) Software becomes more flexible and easily expandable

Cons:

1) Complex implementation (especially with existing code)
2) High degree of dependance at facade interface

THANK YOU FOR LISTENING