DESIGN PATTERN

Là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể.

Tại sao phải sử dụng Design Pattern?

- Giúp sản phẩm của chúng ta linh hoạt, dễ dàng thay đổi và bảo trì hơn.
- Có một điều luôn xảy ra trong phát triển phần mềm, đó là sự thay đổi về yêu cầu. Lúc này hệ thống phình to, các tính năng mới được thêm vào trong khi performance cần được tối ưu hơn.
- Design pattern cung cấp những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Các giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.
- Những lúc khi bạn gặp bất kỳ khó khăn đối với những vấn đề đã được giải quyết rồi, design patterns là hướng đi giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp tốn kém thời gian.
- Giúp cho các lập trình viên có thể hiểu code của người khác một cách nhanh chóng (có thể hiểu là các mối quan hệ giữa các module chẳng hạn). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không tốn nhiều thời gian.

OBSERVER PATTERN

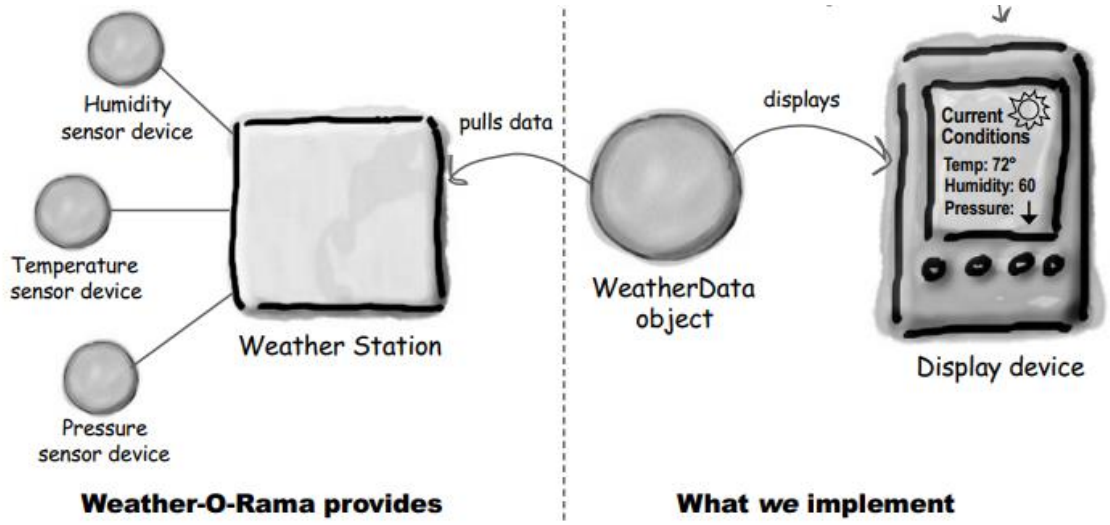
➢ Bài toán cần giải quyết

Giả sử team của bạn vừa có được hợp đồng xây dựng Weather-O-Rama, Trạm giám sát thời tiết dựa trên mạng Internet. Trạm sẽ dựa trên object Weather Data, object này sẽ giám sát tình trạng thời tiết (nhiệt độ, độ ẩm…). Chúng ta cần tạo ứng dụng cung cấp 3 yếu tố: điều kiện hiện tại, thống kê thời tiết và dự báo đơn giản, mọi thứ cần được cập nhật realtime bởi object Weather Data lấy các chỉ số đo lường gần nhất. Hơn thế nữa, trạm thời tiết này cần có thể được mở rộng để các dev khác có thể tự thiết kế thiết bị hiện thị cho riêng họ.

○ Tổng quan ứng dụng theo dõi thời tiết

3 yếu tố của hệ thống:

- Trạm thời tiết
- Weather Data object
- Thiết bị hiển thị



Weather Data object có thể nói chuyện được với trạm thời tiết để lấy dữ liệu, sau đó cập nhật thiết bị hiển thị để hiện các mục: điều khiện hiện tại, dự đoán và thống kê.

Giới thiệu Observer Pattern

- Observer Pattern hoạt động như Youtube hiện nay:

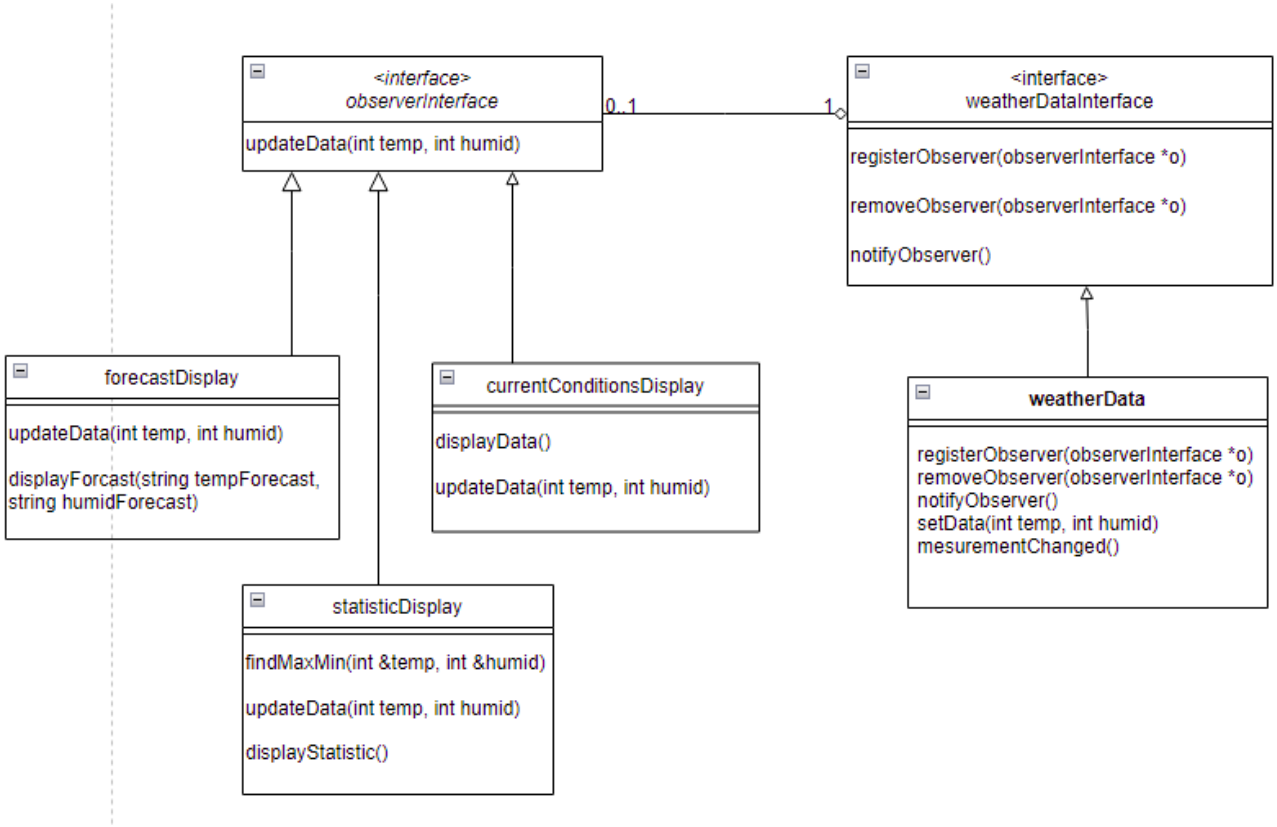
Youtuber mới hoạt động và bắt đầu đăng tải video. Bạn đăng ký kênh Youtube nào đấy, mỗi khi họ đăng video mới thì video đấy sẽ được hiển thị trong tab đăng ký cho bạn. Chỉ khi bạn vẫn còn đăng ký thì video mới được hiển thị trong tab đăng ký cho bạn. Trừ khi kênh Youtube ấy ăn gậy và bị ban thì người ta vẫn còn tiếp tục đăng ký.

Observer Pattern cung cấp một phương thức thiết kế mà subject với observer được kết nối một cách lỏng lẻo (Loose Coupling)

- Subject chỉ biết là Observer áp dụng một interface nào đó
- Có thể thêm bớt Observer mới bất kì lúc nào
- Chúng ta không cần chỉnh sửa subject để thêm kiểu observer mới bởi vì subject sẽ chỉ cung cấp thông báo mới cho observer nào áp dụng Observer interface
- Chúng ta có thể tái sử dụng subject hoặc observer một các độc lập
- Thay đổi phía bên này không ảnh hưởng phía bên kia

Đi vào thiết kế trạm dự báo thời tiết

Diagram của trạm dự báo thời tiết



- Code
- Interface cho observer và weatherData

```
class observerInterface
{
    public:
        virtual void updateData(int temp, int humid) = 0;
};
class weatherDataInterface
{
    public:
        virtual void registerObserver(observerInterface *o) = 0;
        virtual void removeObserver(observerInterface *o) = 0;
        virtual void notifyObserver() = 0;
};
```

- Class weatherData: Class này sẽ làm nhiệm vụ đăng ký, xoá bỏ và thông báo các observer. Vector observerList sẽ lưu trữ các observer đăng ký đến. Việc dữ liệu thay đổi sẽ được mô phỏng bằng method setData();

```
class weatherData : public weatherDataInterface
{
    private:
        int mTemp;
        int mHumid;
        vector <observerInterface *> observerList;
    public:
        void registerObserver(observerInterface *o)
        {
            observerList.push_back(o);
        }
```

```
        void removeObserver(observerInterface *o)
        {
            if(observerList.size() != 0)
            {
                for(int i = 0; i < observerList.size(); i++)
                {
                    if(observerList[i] == o)
                    {
                        observerList.erase(observerList.begin()+i);
                    }
                }
            }
            else
            {
                cout << "no one here" << endl;
            }
        }
        void notifyObserver()
        {
            if(observerList.size() != 0)
            {
                for(int i = 0; i < observerList.size(); i++)
                {
                    observerList[i]->updateData(mTemp, mHumid);
                }
            }
            else
            {
                cout << "no one here" << endl;
            }
        }
        void setData(int temp, int humid)
        {
            mTemp = temp;
            mHumid = humid;
            mesurementChanged();
        }
        void mesurementChanged()
        {
            notifyObserver();
        }
};
```

- Các Class hiển thị sẽ kế thừa observerInterface để tạo ra một hợp đồng chung cho các Class

Ví dụ cho một class hiển thị

```
class currentConditionsDisplay: public observerInterface
{
    private:
        int mTemp;
        int mHumid;
    public:
        currentConditionsDisplay()
        {
```

```
            mTemp = 0;
            mHumid = 0;
        }
        void updateData(int temp, int humid)
        {
            mTemp = temp;
            mHumid = humid;
            displayData();
        }
        void displayData()
        {
            cout << "Current Conditions Display Temprature: " << mTemp << " Humidity: " <<
mHumid << endl;
        }
};
```

- Hàm main() áp dụng trạm thời tiết

```
int main()
{
    weatherData weatherDataObject;
    currentConditionsDisplay CurrentCondition1;
    statisticDisplay statistic1;
    forecastDisplay forecast2;
    weatherDataObject.registerObserver(&CurrentCondition1);
    weatherDataObject.registerObserver(&statistic1);
    weatherDataObject.registerObserver(&forecast2);
    weatherDataObject.setData(100, 20);
    weatherDataObject.removeObserver(&forecast2);
    weatherDataObject.setData(20, 20);
    return 0;
```

- Kết quả chạy:

```
Current Conditions Display Temprature: 100 Humidity: 20
Statistic Display Max/Min Temprature: 100/30 Max/Min Humidity: 50/20
Forecast Display Temprature: warmer Humidity: less moist
Current Conditions Display Temprature: 20 Humidity: 20
Statistic Display Max/Min Temprature: 100/20 Max/Min Humidity: 50/20
```

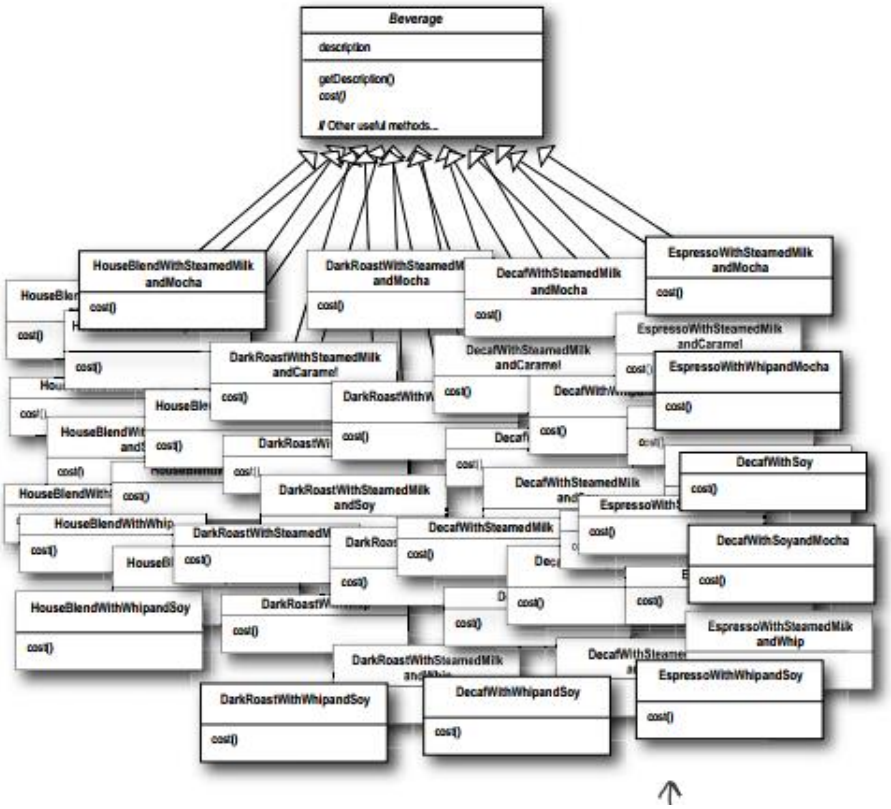
Có thể thấy, ban đầu 3 class display đăng ký thì dữ liệu được cập nhật cho cả 3, nhưng sau khi forecastDisplay huỷ đăng ký thì không còn được thông báo về cập nhật mới nữa

Link thư mục code đầy đủ: [Design-Pattern/WeatherMonitor.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

DECORATOR PATTERN

➢ Bài toán cần giải quyết

Starbuzz Coffee đã trở thành chuỗi buôn bán coffee phát triển nhanh nhất. Bạn có thể thấy hàng quán của họ ở mọi nơi. Bởi vậy họ cần cập nhật hệ thống đặt nước uống của mình để phù hợp với nhu cầu. Không chỉ có coffe mà còn các loại topping khác nữa, và mỗi loại topping thêm thì sẽ có mức giá riêng của nó. Starbuzz cần nghĩ cách áp dụng chúng vào hệ thống đặt hàng của mình



Nếu áp dụng như này thì khi bảo trì thì đúng là một cơn ác mộng

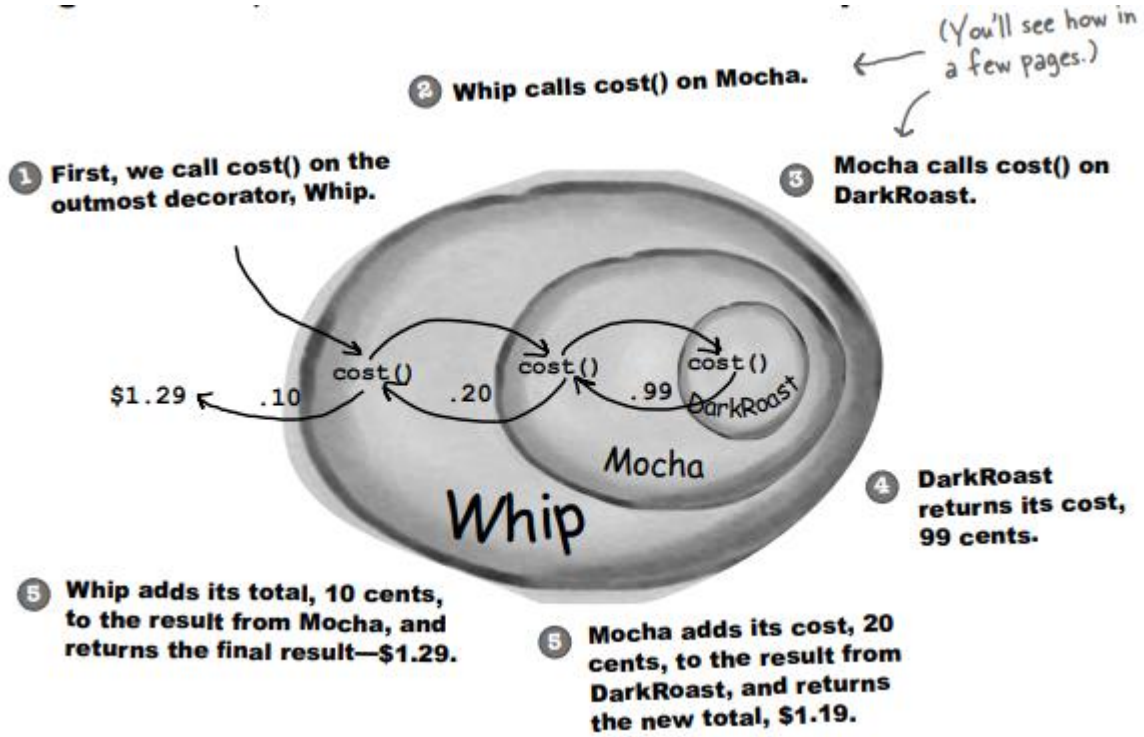
○ Quy tắc Open-closed

Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

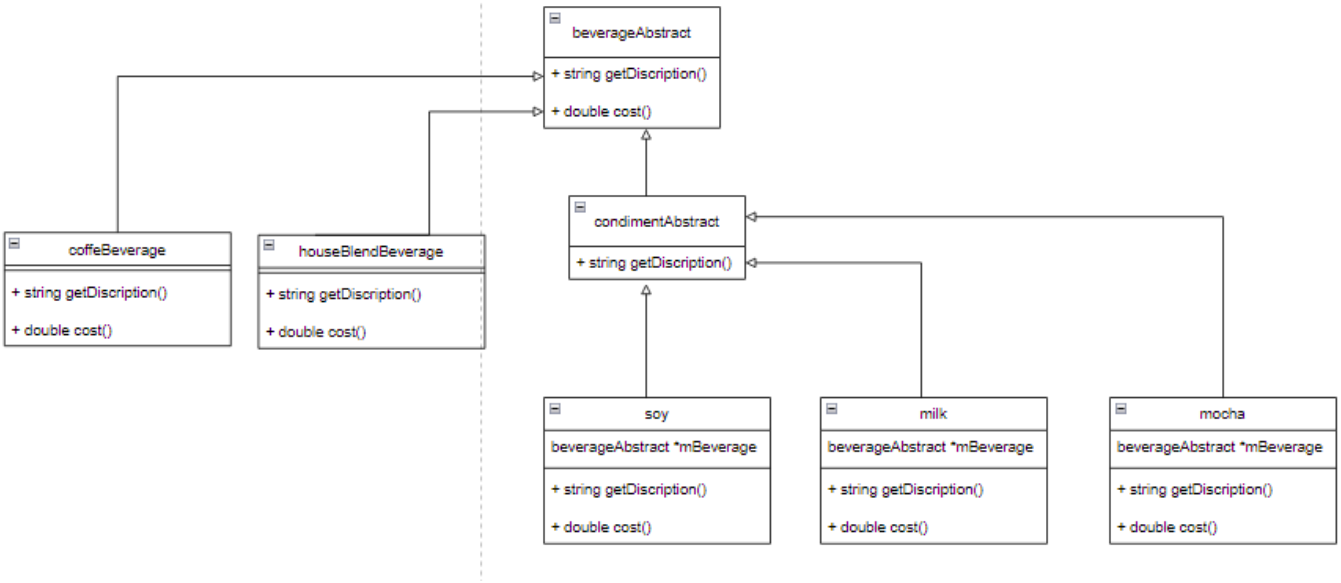
Mục tiêu của quy tắc này là thiết kế làm sao cho các class có thể dễ dàng mở rộng để áp dụng các hành vi mới mà không ảnh hưởng đến code cũ

- Giới thiệu Decorator Pattern



Khởi đầu bằng DarkRoast object, nếu khách gọi Mocha thì ta sẽ tạo ra Mocha class để gói DarkRoast lại. tương tự với các object topping khác. Khi tính tiền, ta sẽ gọi method cost() từ object ngoài cùng và uỷ quyền tính toán đẩy lại cho các object bên trong.

- Đi vào thiết kế của StarBuzz



- Bắt đầu code

Class beverageAbstract chỉ gồm các abstract method, mục đích là để uỷ quyền nhiệm vụ đến các object kế thừa

```
class beverageAbstract
{
    private:
        string discription = "no beverage";
    public:
        virtual string getDiscription()
        {
            return discription;
```

```
        }
        virtual double cost()
        {
            return 0;
        }
};
```

- condimentAbstract class sẽ kế thừa beverageAbstract

```
class condimentAbstract: public beverageAbstract
{
    public:
        virtual string getDiscription()
        {
            return 0;
        }
};
```

- Ví dụ Class cho một loại đồ uống chính:

```
class houseBlendBeverage: public beverageAbstract
{
    public:
        string getDiscription()
        {
            return "House Blend";
        }
        double cost()
        {
            return 2.5;
        }
};
```

- Ví dụ Class cho một loại topping

```
class milk: public condimentAbstract
{
    public:
    beverageAbstract *mBeverage;
        milk(beverageAbstract *beverage)
        {
            mBeverage = beverage;
        }
        string getDiscription()
        {
            return  mBeverage->getDiscription() + " " + "milk";
        }
        double cost()
        {
            return mBeverage->cost() + 2.5;
        }
};
```

- Áp dụng trong hàm main()

```
int main()
{
    beverageAbstract *beveragePtr;
    beveragePtr = new houseBlendBeverage;
    beveragePtr = new milk(beveragePtr);
    beveragePtr = new soy(beveragePtr);
```

```
    cout << beveragePtr->getDiscription() << " " << beveragePtr->cost() << endl;
    cout << "oder new drink " << endl;
    beveragePtr = new coffeBeverage;
    beveragePtr = new milk(beveragePtr);
    cout << beveragePtr->getDiscription() << " " << beveragePtr->cost() << endl;
}
```

- Kết quả chạy:

```
House Blend milk Soy 10.5
oder new drink
Coffe milk 6
```

Link thư mục code đầy đủ: [Design-Pattern/Starbuzz.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

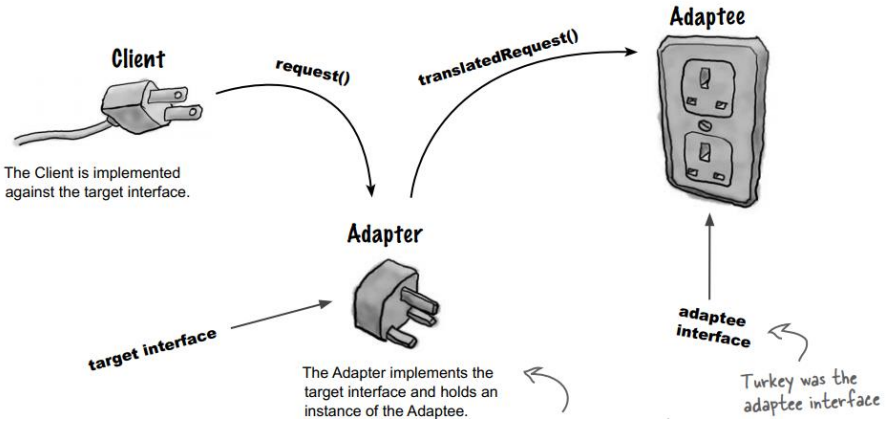
ADAPTER PATTERN

➢ Bài toán đặt ra

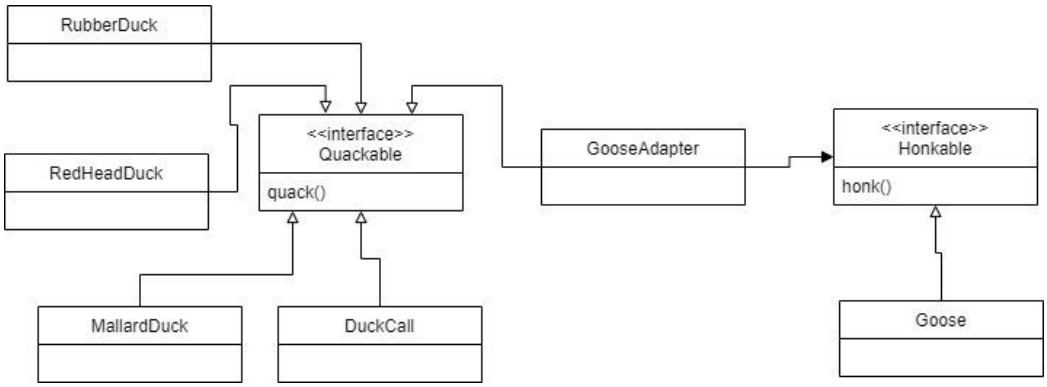
Giả sử bạn đang phát triển một máy mô phỏng vịt kêu quack(). Quá trình phát triển đã đến hồi kết thì bỗng nhiên bên đầu tư muốn mô phỏng thêm ngan kêu honk(). Giờ đập đi xây lại rất mất công nên bạn muốn tìm một cách nào đó mà vẫn mô phỏng được ngan mà không phải làm lại bất cứ thứ gì.

➢ Giới thiệu Adapter pattern

Adapter pattern chuyển đổi interface của một class thành một interface khác mà khách hàng mong đợi. Adapter pattern cho phép các lớp hoạt động cùng nhau mà mặc dù trước đó không thể vì interface không tương thích



➢ Lưu đồ



➢ Code

Interface cho quack() và honk()

```
class Quackable
{
    public:
        virtual void quack() = 0;
};
class Honkable
{
    public:
        virtual void honk() = 0;
};
```

Class cho 1 loại vịt

```
class MallardDuck : public Quackable
{
    public:
        void quack()
        {
```

```
            cout << "MallardDuck quack" << endl;
        }
};
```

Class cho 1 loại ngan

```
class Goose : public Honkable
{
    public:
        void honk()
        {
            cout << "Goose Honk" << endl;
        }
};
```

Adapter để chuyển honk() sang quack()

```
class GooseAdapter : public Quackable
{
    Honkable *mGoose;
    public:
        GooseAdapter(Honkable *goose) : mGoose(goose) {}
        void quack()
        {
            mGoose -> honk();
        }
};
```

Main()

```
void simulate(Quackable *duck)
{
    duck->quack();
}
int main()
{
    Quackable* duck1 = new MallardDuck();
    Quackable* duck2 = new RedHeadDuck();
    Quackable* duck3 = new DuckCall();
    Quackable* duck4 = new RubberDuck();
    Honkable* goose1 = new Goose();
    Quackable* gooseDuck1 = new GooseAdapter(goose1);
    simulate(duck1);
    simulate(duck2);
    simulate(duck3);
    simulate(duck4);
    simulate(gooseDuck1);
    return 0;
}
```

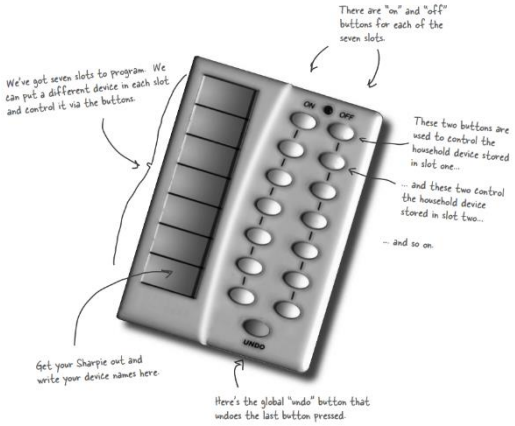
Kết quả chạy:

```
MallardDuck quack
RedHeadDuck quack
DuckCall kwak
RubberDuck squeak
Goose Honk
```

Link code đầy đủ: [Design-Pattern/DuckAdapter.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

COMMAND PATTERN

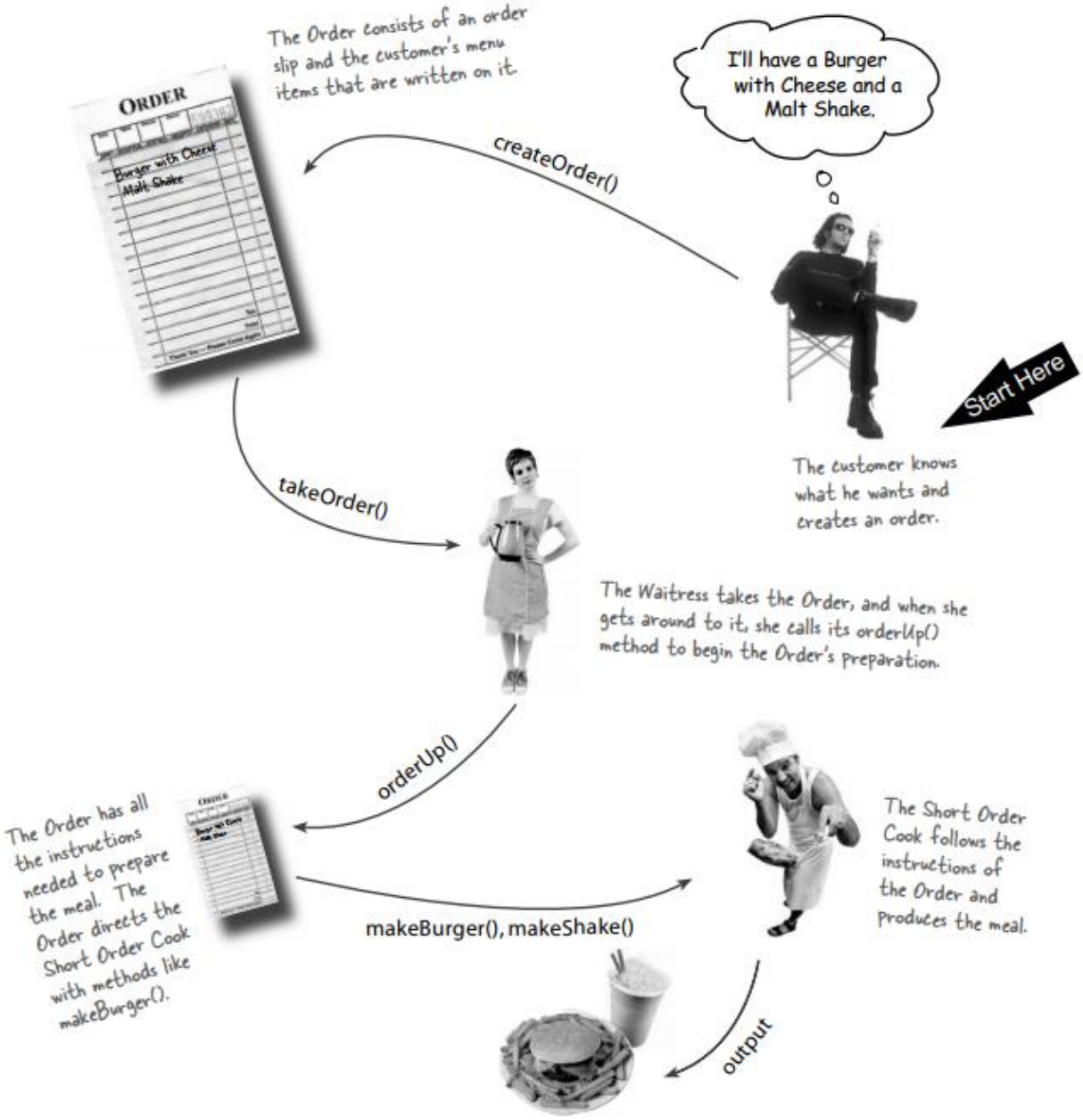
➢ Bài toán cần giải quyết



Bạn cần thiết kế một cái điều khiểu mà có 7 slot có thể thiết đặt, mỗi thiết đặt có thể gán với các thiết bị trong nhà cùng với các nút on off cho mỗi slot. Điều khiển này cũng có một nút undo. Các thiết bị có thể từ các nhà sản xuất khác nhau. API cần có thể điều khiển được các thiết bị mà nhà sản xuất có thể tung ra trong tương lai.

➢ Giới thiệu Command Pattern

Có thể hiểu Command Pattern giống như một nhà hàng đang hoạt động. Khác hàng gọi món createOrder(), món sẽ được thêm vào tờ Order, sau đó phục vụ sẽ nhận tờ order takeOrder(). Khi đến đơn của bạn thì phục vụ sẽ gọi orderUp() để đầu bếp bắt đầu chuẩn bị đơn, trong tờ đơn sẽ có chi tiết các phương thức để tạo ra món được đặt.

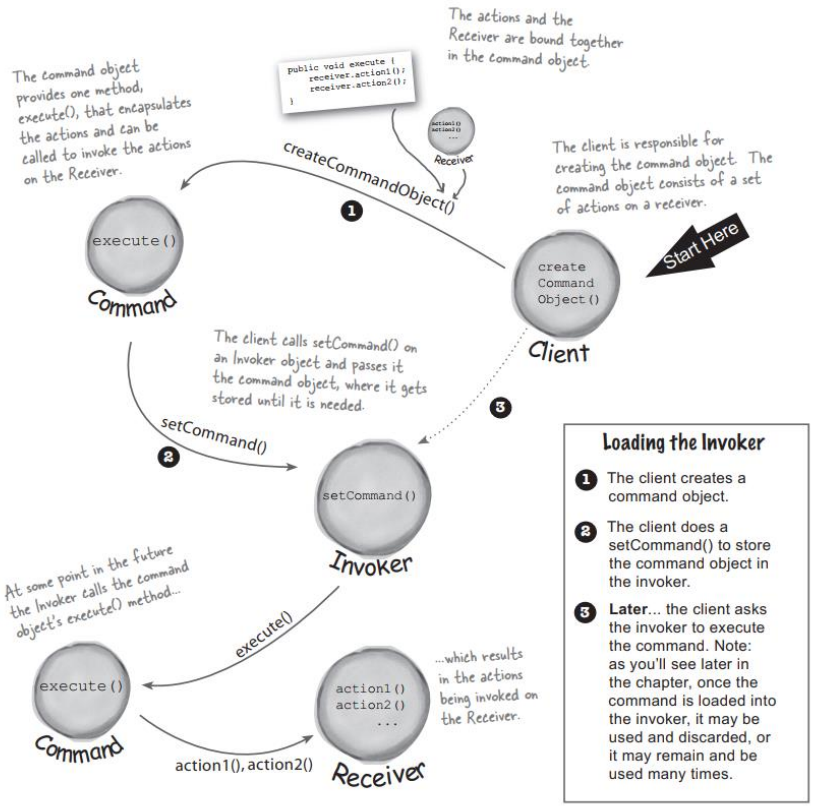


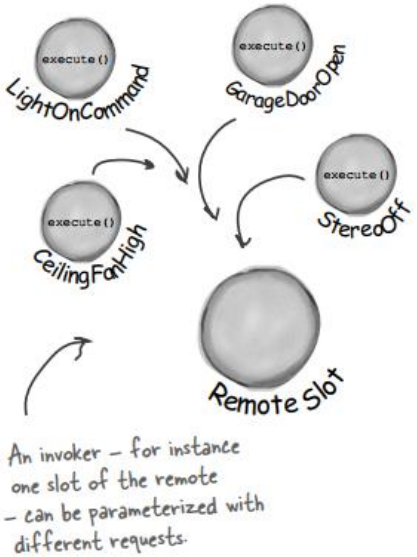
- Tờ Order đóng gói yêu cầu đặt món, giống như 1 object, nó có thể được chuyển giao qua lại, từ phục vụ đến bàn order đến các phục vụ khác. Nó chỉ có 1 method duy nhất oderUp() mà đóng gói các hoạt động để chuẩn bị đơn. Và nó cũng tham chiếu đến object đầu bếp.

- Phục vụ chỉ có 1 nhiệm vụ duy nhất đó là tiếp tục nhận đơn cho đến khi đến được quầy order và gọi oderUp() để đầu bếp chuẩn bị món ăn.

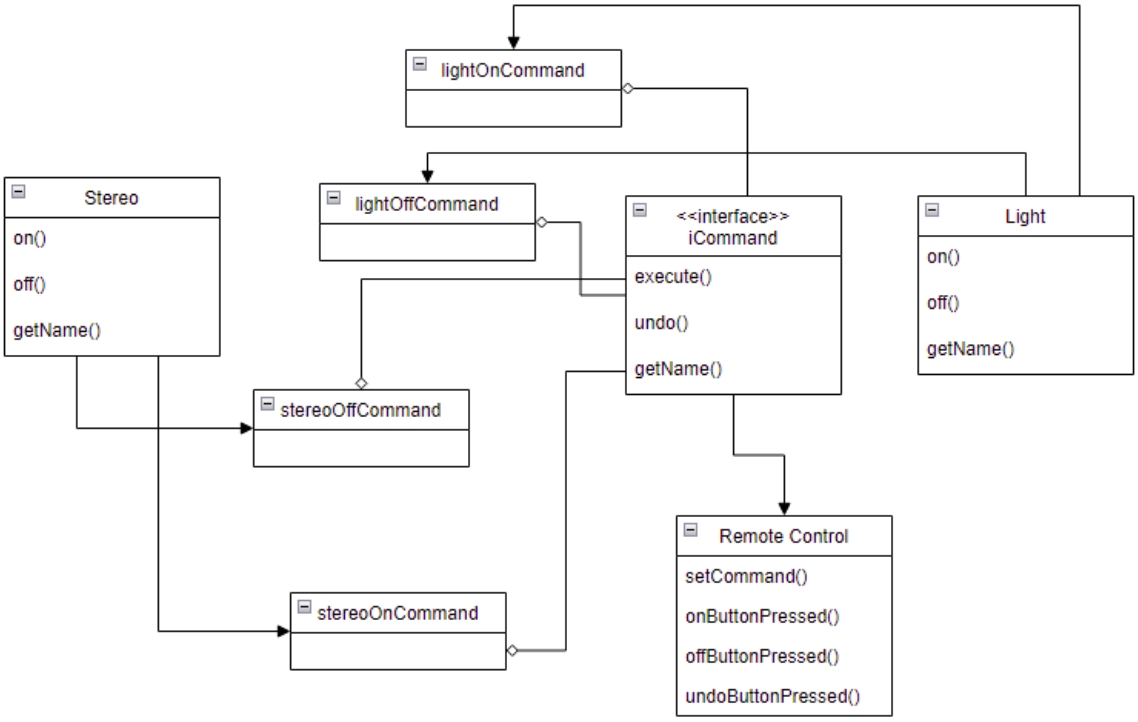
- Đầu bếp là thành phần duy nhất biết nấu ăn. Khi phục vụ gọi orderUp() thì đầu bếp áp dụng các method để nấu. Đầu bếp và phục vụ hoàn toàn tách rời, phục vụ có tờ Order đóng gói tất các chi tiết của món ăn, chỉ cần gọi 1 method trong mỗi đơn để nó đc đầu bếp chuẩn bị. Còn đầu bếp thì lấy các chỉ dẫn từ tờ order, cả 2 không cần trực tiếp giao tiếp với nhau.

- Ta có thể liên tưởng đến Command Pattern giống như ví dụ trên:
- Client tạo CommandObject có list các lệnh cần thực hiện, CommandObject có 1 method execute() đóng gói để thực hiện các lệnh trên. Sau đó client gọi SetCommand tại Invoker để đưa nó CommandObject, CommandObject được lưu cho đến khi cần, giống như khách hàng đưa phục vụ tờ Order
- Sau đó Invoker gọi execute để Reciver thực hiện các lệnh. Giống như phục vụ pass tờ order cho đầu bếp để bắt đầu nấu





- Lưu đồ Remote



- Code
  - Class iCommand là 1 interface

```
class iCommand
{
    public:
        virtual void execute() = 0;
        virtual void undo() = 0;
        virtual string getName() = 0;
};
```

  - Ví dụ cho 1 class đồ dùng trong nhà

mName được truyền vào tương ứng với các vị trí trong nhà, ví dụ Kitchen Light

```
class Light
{
    private:
        string mName;
    public:
        Light(string name)
        {
            mName = name;
        }
        void on()
        {
            cout << mName << " light on" << endl;
        }
        void off()
        {
            cout << mName << " light off" << endl;
        }
        string getName()
```

```
        {
            return mName;
        }
};
```

- Command để thực hiện bật đèn

```
class lightOnCommand : public iCommand
{
    private:
        Light *mLight;
    public:
        lightOnCommand(Light *Light)
        {
            mLight = Light;
        }
        void execute()
        {
            mLight->on();
        }
        void undo()
        {
            mLight->off();
        }
        string getName()
        {
            return "lightOnCommand";
        }
};
```

- Class cho RemoteControl, class này được khởi tạo cùng với số lượng Command cần ghi.

Ban đầu các slot sẽ được khởi tạo bởi object noCommand để tránh khỏi việc phải check mỗi lần nhấn nút. Các method của object này không có tác dụng gì cả

```
class RemoteControl
{
private:
    int mMaxCommand;
    vector <iCommand *> mOnCommand;
    vector <iCommand *> mOffCommand;
    iCommand *mUndoCommand;
public:
    RemoteControl(int maxCommand)
    {
        mMaxCommand = maxCommand;
        iCommand *noCommand = new(NoCommand);
        for (int i = 0; i < mMaxCommand; i++)
        {
            mOnCommand.push_back(noCommand);
            mOffCommand.push_back(noCommand);
        }
        mUndoCommand = noCommand;
    }
    ~RemoteControl() {}
```

```
    void setCommand(int slot, iCommand *onCommand, iCommand *offCommand)
    {
        mOnCommand[slot] = onCommand;
        mOffCommand[slot] = offCommand;
    }
    void onButtonPressed(int slot)
    {
        mOnCommand[slot]->execute();
        mUndoCommand = mOnCommand[slot];
    }
    void offButtonPressed(int slot)
    {
        mOffCommand[slot]->execute();
        mUndoCommand = mOffCommand[slot];
    }
    void undoButtonPressed()
    {
        mUndoCommand->undo();
    }
    void getList()
    {
        for (int i = 0; i < mOnCommand.capacity(); i++)
        {
            cout << "slot:" << i << ": " << mOnCommand[i]->getName() << " " <<
mOffCommand[i]->getName() << endl;
        }
    }
};
```

- Hàm main()

```
int main()

{
    RemoteControl remote1(4);
    Light light1("kitchen");
    Light light2("toilet");
    Stereo stereo1("living");
    Stereo stereo2("bedroom");

    lightOnCommand lightOnCommand1(&light1);
    lightOffCommand lightOffCommand1(&light1);
    lightOnCommand lightOnCommand2(&light2);
    lightOffCommand lightOffCommand2(&light2);
    stereoOnCommand stereoOnCommand1(&stereo1);
    stereoOffCommand stereoOffCommand1(&stereo1);
    stereoOnCommand stereoOnCommand2(&stereo2);
    stereoOffCommand stereoOffCommand2(&stereo2);

    remote1.setCommand(0, &lightOnCommand1, &lightOffCommand1);
    remote1.setCommand(1, &lightOnCommand2, &lightOffCommand2);
    remote1.setCommand(2, &stereoOnCommand1, &stereoOffCommand1);
    remote1.setCommand(3, &stereoOnCommand2, &stereoOffCommand2);
    remote1.getList();
```

```
    remote1.onButtonPressed(0);
    remote1.onButtonPressed(1);
    remote1.onButtonPressed(2);
    remote1.onButtonPressed(3);
    remote1.undoButtonPressed();
    remote1.offButtonPressed(0);
    remote1.offButtonPressed(1);
    remote1.offButtonPressed(2);
    remote1.offButtonPressed(3);
}
```

- ✓ Kết quả chạy:

```
slot:0: lightOnCommand lightOffCommand
slot:1: lightOnCommand lightOffCommand
slot:2: stereoOnCommand stereoOffCommand
slot:3: stereoOnCommand stereoOffCommand
kitchen light on
toilet light on
living stereo on
bedroom stereo on
bedroom stereo off
kitchen light off
toilet light off
living stereo off
bedroom stereo off
```

Link code đầy đủ: [Design-Pattern/Remote.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

SINGLETON PATTERN

➢ Bài toán cần giải quyết

Giả sử có một nhà máy sản suất chocolate sử dụng máy tính để điều khiển máy nung chocolate. Nhiệm vụ của máy nung là nhận chocolate và sữa, nung sôi và đưa chúng vào giai đoạn tiếp theo để tạo ra thanh chocolate. Nhiệm vụ của bạn là thiết kế làm sao chỉ có duy nhất 1 ChocolateBoiler được khởi tạo, bởi nếu xuất hiện 2 cái thì tình huống sẽ khó lường

➢ Code

- Class cho ChocolateBoiler

```
class ChocolateBoiler
{
private:
    bool mEmpty;
    bool mBoiled;
    string mName;
    static mutex mLocker;
    static ChocolateBoiler *mInstance;
    ChocolateBoiler(string name)
    {
        mName = name;
        mEmpty = true;
        mBoiled = false;
    }
public:
    void fill()
    {
        if (isEmpty())
        {
            mEmpty = false;
            mBoiled = false;
            //fill
        }
    }
    void drain()
    {
        if(!isEmpty() && isBoiled())
        {
            //drain
            mEmpty = true;
        }
    }
    void boil()
    {
        if(!isEmpty() && !isBoiled())
        {
            //boil
            mBoiled = true;
        }
    }
    bool isEmpty()
    {
```

```
        return mEmpty;
    }
    bool isBoiled()
    {
        return mBoiled;
    }
    string getName()
    {
        return mName;
    }
    static ChocolateBoiler *getInstance(string name)
    {
        if(mInstance == nullptr)
        {
            mInstance = new ChocolateBoiler(name);
        }
        return mInstance;
    }
};
```

- Constructor sẽ được cho vào private để không ai bên ngoài có thể khởi tạo được nó

```
    ChocolateBoiler(string name)
    {
        mName = name;
        mEmpty = true;
        mBoiled = false;
    }
```

- Method getInstance được sử dụng để khởi tạo object

```
    static ChocolateBoiler *getInstance(string name)
    {
        if(mInstance == nullptr)
        {
            mInstance = new ChocolateBoiler(name);
        }
        return mInstance;
    }
```

- Con trỏ static mInstance sẽ được khởi tạo nullptr

```
ChocolateBoiler* ChocolateBoiler :: mInstance = nullptr;
```

- Hàm main()

```
int main()
{
    ChocolateBoiler *chocolate1;
    chocolate1 = ChocolateBoiler::getInstance("chocolate 1");
    cout << chocolate1->getName() << endl;
    chocolate1 = ChocolateBoiler::getInstance("chocolate 2");
    cout << chocolate1->getName() << endl;
}
```

- Kết quả chạy:

Có thể thấy chỉ duy nhất 1 object chocolate1 được khởi tạo

```
chocolate 1
chocolate 1
```

➢ Vấn đề multithreading

Có một lỗ hổng tiềm ẩn nguy hiểm ở đây. Chúng ta muốn đảm bảo rằng chỉ có duy nhất 1 object của class ChocolateBoiler có thể được tạo ra. Nhưng nếu chương trình có nhiều threads cùng chạy một lúc và cùng call đến method getInstance() tại cùng thời điểm, chúng ta có thể gặp rắc rối ở đây. Lỗ hổng ở đây nằm ở câu lệnh check null con trỏ mInstancePtr.

✓ Cách giải quyết

Thêm mutex lock cho chương trình

```
static ChocolateBoiler *getInstance(string name)
{
    mLocker.lock();
    if(mInstance == nullptr)
    {
        mInstance = new ChocolateBoiler(name);
    }
    mLocker.unlock();
    return mInstance;
}
```

Khởi tạo object ngay từ ban đầu

```
ChocolateBoiler* ChocolateBoiler :: mInstance = new ChocolateBoiler("chocolate 0");
```

Link code đầy đủ: [Design-Pattern/ChocolateFactory.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

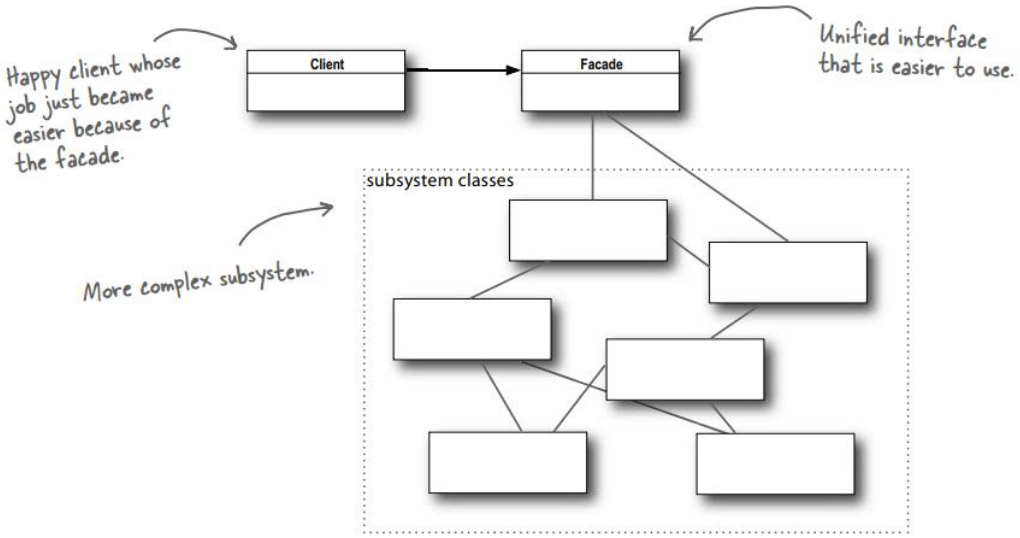
FACADE PATTERN

➢ Bài toán cần giải quyết

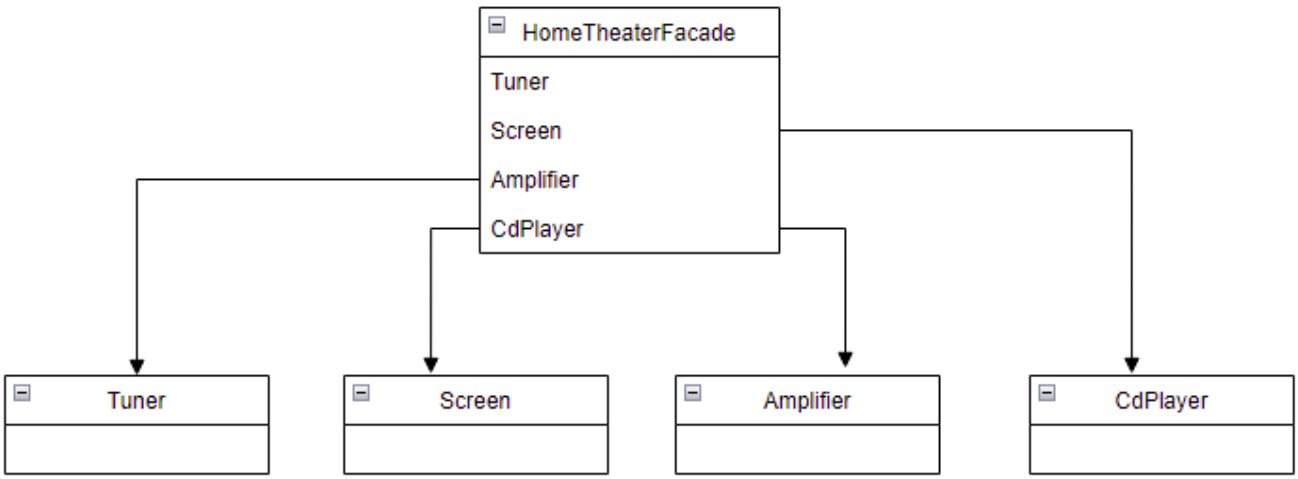
Bạn cần xây dựng rạp phim tại gia riêng cho bản thân, bạn muốn thiết kế quy trình vận hành thật đơn giản để ai trong gia đình cũng thực hiện được

➢ Giới thiệu Facade Pattern

Với Façade Pattern, chúng ta tạo một class cấp cao mà đơn giản hoá và thống nhất các class phức tạp khác. Client sau đó chỉ việc tương tác với class cấp cao đó



➢ Lưu đồ



- Code

Ví dụ cho một Class cho một tính năng phụ

```
class Amplifier
{
    public:
        void on()
        {
            cout << "Ampli on" << endl;
        }
        void off()
        {
            cout << "Ampli on" << endl;
        }
        void setStereoSound()
        {
            cout << "set Stereo Sound" << endl;
        }
        void setSurroundSound()
        {
            cout << "set Surround Sound" << endl;
        }
        void setVolume()
        {
            cout << "set Volume" << endl;
        }
};
```

Class chính sẽ thống nhất và đơn giản hoá các tính năng

```
class HomeTheaterFacade
{
    Tuner* mTuner;
    CdPlayer* mCdPlayer;
    Amplifier* mAmpli;
    Screen* mScreen;
    public:
        HomeTheaterFacade(Tuner* tuner, CdPlayer* cdPlayer, Amplifier* ampli, Screen*
screen)
        {
            mTuner = tuner;
            mCdPlayer = cdPlayer;
            mAmpli = ampli;
            mScreen = screen;
        }
        void doThing1()
        {
            cout << "doThing1" << endl;
            mTuner->on();
            mCdPlayer->pause();
            mScreen->up();
        }
        void doThing2()
        {
            cout << "doThing2" << endl;
            mTuner->setAm();
```

```
            mCdPlayer->eject();
            mScreen->down();
            mAmpli->setStereoSound();
        }
        void doThing3()
        {
            cout << "doThing3" << endl;
            mTuner->setFm();
            mScreen->up();
            mCdPlayer->play();
        }
};
```

Hàm main() áp dụng các tính năng trên

```
int main()
{
    Tuner tuner1;
    CdPlayer cdPlayer1;
    Amplifier ampli1;
    Screen screen1;
    HomeTheaterFacade homeTheater1(&tuner1, &cdPlayer1, &ampli1, &screen1);
    homeTheater1.doThing1();
    homeTheater1.doThing2();
    homeTheater1.doThing3();
    return 0;
}
```

- Kết quả chạy

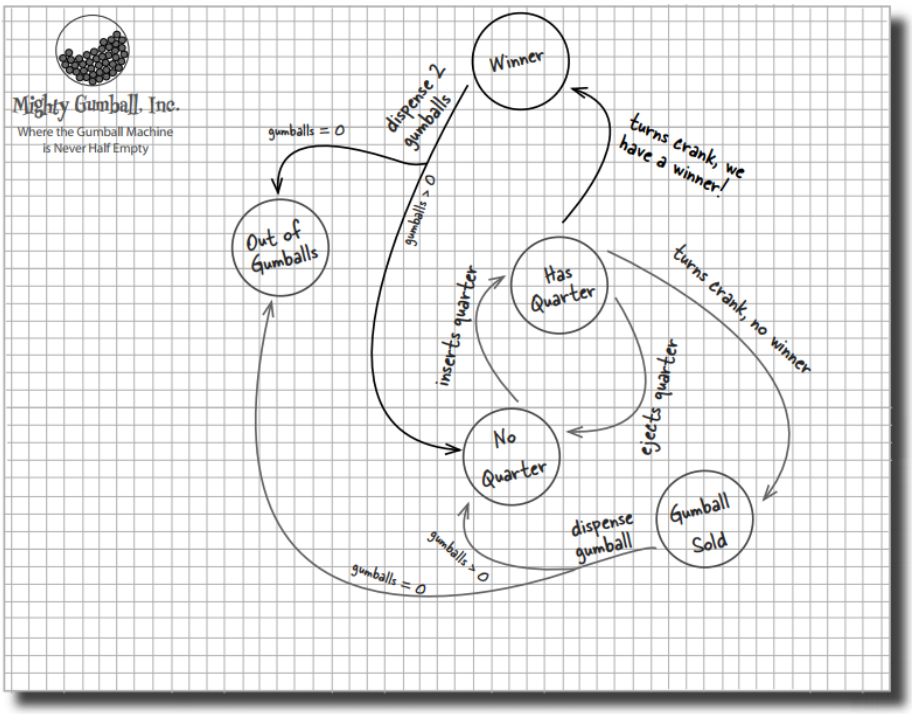
```
doThing1
Tuner on
CD pause
screen up
doThing2
setAm
CD eject
screen down
set Stereo Sound
doThing3
setFm
screen up
CD play
```

Link code đầy đủ: [Design-Pattern/Facade.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

STATE PATTERN

➢ Bài toán cần giải quyết

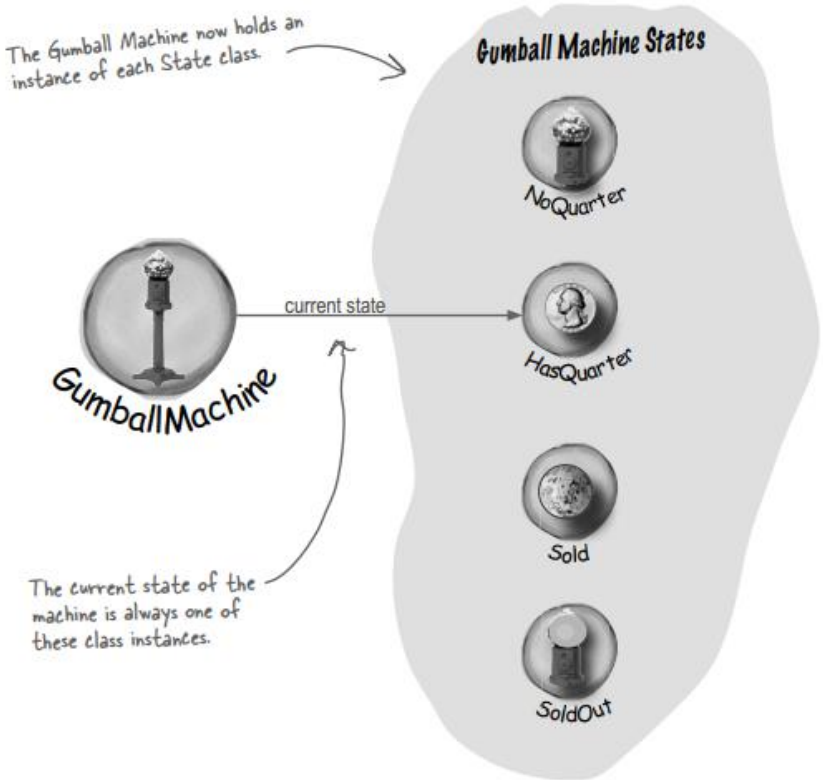
Ngày nay mọi người đang thông minh hoá các thiết bị ngoài đời, như máy nhả kẹo cao su. Các nhà sản xuất lớn nhận thấy rằng bằng cách đưa CPU vào máy của họ, họ có thể tăng doanh số bán hàng, theo dõi hàng tồn kho qua mạng và đo lường mức độ hài lòng của khách hàng chính xác hơn. Máy nhả kẹo có quy trình hoạt động như sau.

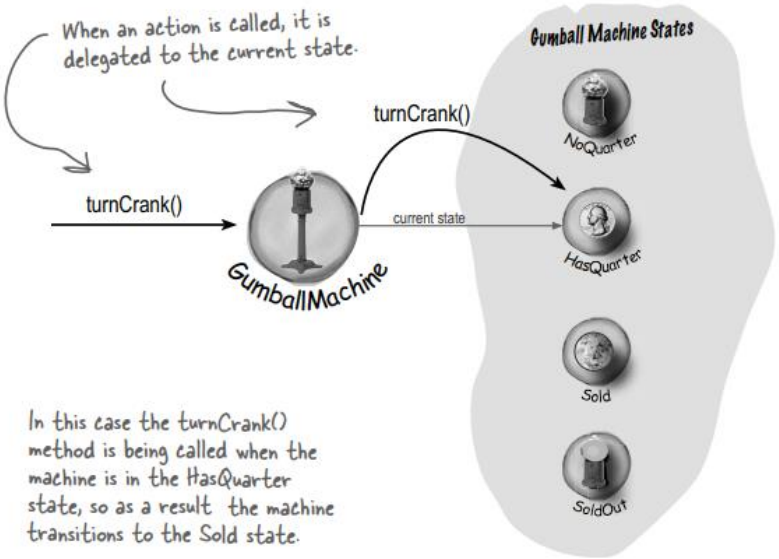


➢ Giới thiệu State Pattern

State Pattern cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi. Bởi vì pattern này đóng gói trạng thái thành các class riêng biệt và ủy quyền cho object đại diện cho trạng thái hiện tại thực hiện các hoạt động.

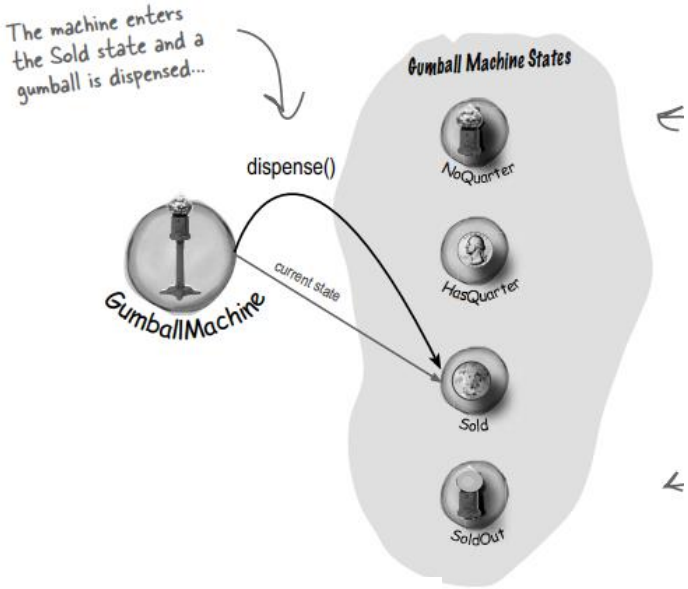
- Gumball Machine sẽ giữ các biến khởi tạo của các Class. Trạng thái hiện tại của Machine sẽ là một trong các biến đấy.



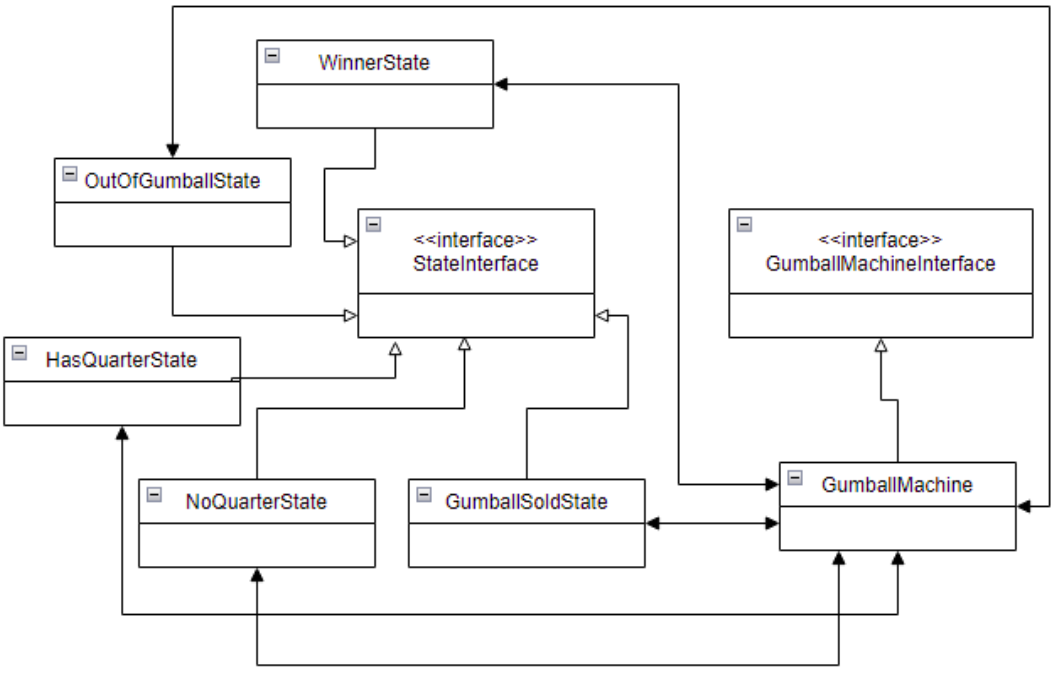


- Khi một hành động được thực hiện thì nó sẽ uỷ quyền đến class trạng thái hiện tại
- Trong trường hợp này thì sẽ hành động turnCrank sẽ được uỷ quyền đến cho trạng thái HasQuarter bởi vì bạn chỉ được gạt cần khi đã có xu trong máy

- Sau đó máy sẽ nhảy đến trạng thái tiếp theo. Trong trường hợp này sẽ là nhả bóng ở trạng



➢ Lưu đồ



- Code

Interface cho các State

```
class StateInterface
{
    public:
        virtual void insertQuarter() = 0;
        virtual void ejectQuarter() = 0;
        virtual void turnCrank() = 0;
        virtual void dispenseGumball() = 0;
};
```

Interface cho máy Gumball

```
class GumballMachineInterface
{
    public:
        virtual void insertQuarter() = 0;
        virtual void ejectQuarter() = 0;
        virtual void turnCrank() = 0;
        virtual void dispenseGumball() = 0;
        virtual StateInterface* getState() = 0;
        virtual void setState(StateInterface* state) = 0;
        virtual StateInterface* getNoQuarterState() = 0;
        virtual StateInterface* getHasQuarterState() = 0;
        virtual StateInterface* getGumballSoldState() = 0;
        virtual StateInterface* getOutOfGumballState() = 0;
        virtual StateInterface* getWinnerState() = 0;
        virtual int getGumballs() = 0;
};
```

Ví dụ 1 state của máy Gumball

```
class WinnerState : public StateInterface
{
    GumballMachineInterface *mGumballMachine;
    public:
        WinnerState(GumballMachineInterface *gumballMachine)
        {
            mGumballMachine = gumballMachine;
        }
        void insertQuarter()
        {
            cout << "Already have one" << endl;
        }
        void ejectQuarter()
        {
            cout << "cant" << endl;
        }
        void turnCrank()
        {
            cout << "can't" << endl;
        }
        void dispenseGumball()
        {
            cout << "holy shit u won" << endl;
```

```
            cout << "here ur gumball" << endl;
            mGumballMachine->dispenseGumball();
            if(mGumballMachine->getGumballs() > 0)
            {
                cout << "here ur gumball" << endl;
                mGumballMachine->dispenseGumball();
                if(mGumballMachine->getGumballs() > 0)
                {
                    mGumballMachine->setState(mGumballMachine->getNoQuarterState());
                }
                else
                {
                    mGumballMachine->setState(mGumballMachine->getOutOfGumballState());
                }
            }
            else
            {
                mGumballMachine->setState(mGumballMachine->getOutOfGumballState());
            }

        }
};
```

Máy Gumball

```
class GumballMachine : public GumballMachineInterface
{
    int mGumballs;
    StateInterface* mCurrentState;
    StateInterface* mHasQuarterState;
    StateInterface* mNoQuarterState;
    StateInterface* mGumballSoldState;
    StateInterface* mOutOfGumballState;
    StateInterface* mWinnerState;
    public:
        GumballMachine(int gumballs)
        {
            mGumballs = gumballs;
            mHasQuarterState = new HasQuarterState(this);
            mNoQuarterState = new NoQuarterState(this);
            mGumballSoldState = new GumballSoldState(this);
            mOutOfGumballState = new OutOfGumballState(this);
            mWinnerState = new WinnerState(this);
            mCurrentState = mNoQuarterState;
        }
        void insertQuarter()
        {
            mCurrentState->insertQuarter();
        }
        void ejectQuarter()
        {
            mCurrentState->ejectQuarter();
        }
        void turnCrank()
        {
```

```
            mCurrentState->turnCrank();
            mCurrentState->dispenseGumball();
        }
        void dispenseGumball()
        {
            if(mGumballs > 0)
            {
                mGumballs--;
            }

        }
        StateInterface* getCurrentState()
        {
            return mCurrentState;
        }
        StateInterface* getNoQuarterState()
        {
            return mNoQuarterState;
        }
        StateInterface* getHasQuarterState()
        {
            return mHasQuarterState;
        }
        StateInterface* getGumballSoldState()
        {
            return mGumballSoldState;
        }
        StateInterface* getOutOfGumballState()
        {
            return mOutOfGumballState;
        }
        StateInterface* getWinnerState()
        {
            return mWinnerState;
        }
        void setState(StateInterface* state)
        {
            mCurrentState = state;
        }
        int getGumballs()
        {
            return mGumballs;
        }
        StateInterface* getState()
        {
            return mCurrentState;
        }
};
```

Hàm main()

```
int main()
{
    int count = 5;
    GumballMachine gumballMachine1(100);
```

```
    while(count > 0)
    {
        cout << "time : " << count << endl;
        gumballMachine1.insertQuarter();
        gumballMachine1.turnCrank();
        count--;
    }
    return 0;
}
```

Kết quả chạy:

```
time : 5
ok got ur quarter
ok turn crank
here ur gumball
time : 4
ok got ur quarter
ok turn crank
here ur gumball
time : 3
ok got ur quarter
ok turn crank
here ur gumball
time : 2
ok got ur quarter
ok turn crank
here ur gumball
time : 1
ok got ur quarter
ok turn crank
here ur gumball
```

Hơi đen vì không trúng thưởng lần nào

Link code đầy đủ: Design-Pattern/Gumball.cpp at master · truongvanhuy2000/Design-Pattern (github.com)

FACTORY PATTERN

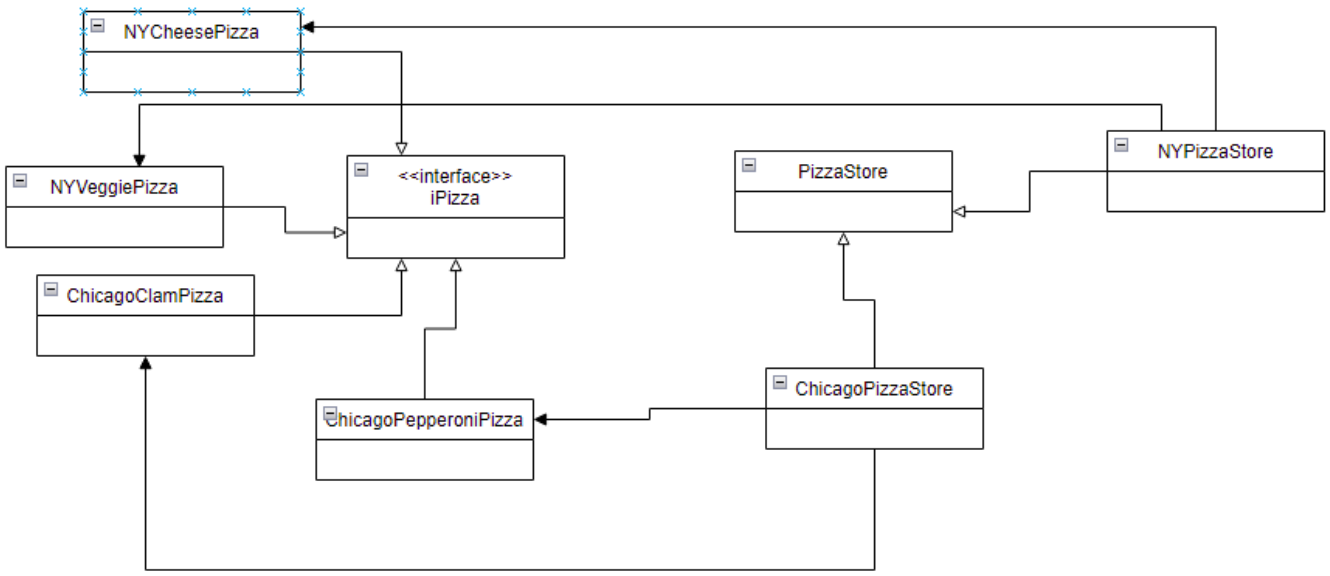
➢ Bài toán cần giải quyết

Bạn cần xây dựng chuỗi bán pizza, các đối thủ của bạn liên tục thêm các loại pizza hợp trend như là Clam pizza, Veggie Pizza. Bạn muốn bắt kịp họ nên bạn sẽ phải thêm các loại pizza mới liên tục vào Menu của mình, và bạn cũng muốn các chuỗi hàng của mình có cách chế biến và hương vị riêng của mình nhưng vẫn có thể kiểm soát được chất lượng đầu ra.

➢ Giới thiệu Factory Pattern

Factory Pattern định nghĩa 1 interface để tạo object, nhưng để class con quyết định xem class nào sẽ được khởi tạo. Factory Method để cho class chuyển quá trình khởi tạo cho class con.

➢ Lưu đồ



➢ Code

Interface cho class Pizza

```
class iPizza
{
    public:
        virtual void prepare() = 0;
        virtual void bake() = 0;
        virtual void cut() = 0;
        virtual void box() = 0;
};
```

Class Pizza Store

```
class PizzaStore
{
    public:
        PizzaStore(){};
        iPizza* oderPizza(string pizzaType)
        {
            iPizza* pizza;
            pizza = createPizza(pizzaType);
            pizza->prepare();
            pizza->bake();
            pizza->cut();
```

```
            pizza->box();
            return pizza;
        }
        virtual iPizza* createPizza(string pizzaType) = 0;
};
```

Có thể thấy, loại Pizza muốn tạo thì sẽ phụ thuộc vào classs cửa hàng con tại từng vùng miền, nhưng mà quy trình chế biến và đóng gói thì sẽ do class chính quy định để đảm bảo chất lượng.

Class ví dụ cho 1 loại Pizza

```
class NYCheesePizza : public iPizza
{
    public:
        void prepare()
        {
            cout << "prepare NYCheesePizza" << endl;
        }
        void bake()
        {
            cout << "bake NYCheesePizza" << endl;
        }
        void cut()
        {
            cout << "cut NYCheesePizza" << endl;
        }
        void box()
        {
            cout << "box NYCheesePizza" << endl;
        }
};
```

Class mẫu cho 1 cửa hàng Pizza

```
class ChicagoPizzaStore : public PizzaStore
{
    public:
        iPizza* createPizza(string pizzaType)
        {
            if(pizzaType == "ChicagoPepperoniPizza")
            {
                return new ChicagoPepperoniPizza();
            }
            else if(pizzaType == "ChicagoClamPizza")
            {
                return new ChicagoClamPizza();
            }
            else
            return nullptr;
        }
};
```

Hàm main()

```
int main()
{
    PizzaStore* nyPizzaStore = new NYPizzaStore();
    nyPizzaStore->oderPizza("NYVeggiePizza");
    PizzaStore* chicagoPizzaStore = new ChicagoPizzaStore();
    chicagoPizzaStore->oderPizza("ChicagoPepperoniPizza");
```

```
    return 0;
}
```

- Kết quả chạy:

```
prepare NYVeggiePizza
bake NYVeggiePizza
cut NYVeggiePizza
box NYVeggiePizza
prepare ChicagoPepperoniPizza
bake ChicagoPepperoniPizza
cut ChicagoPepperoniPizza
box ChicagoPepperoniPizza
```

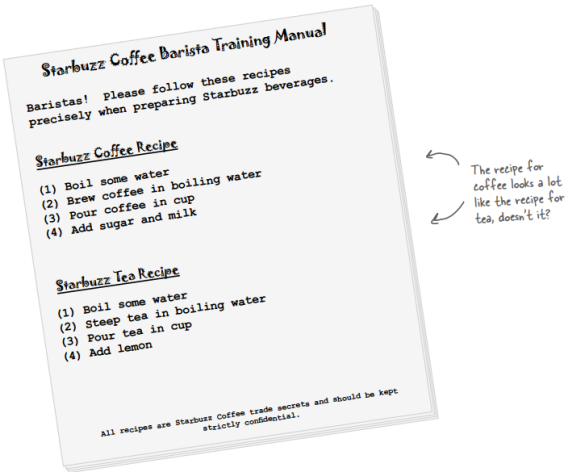
Link code đầy đủ: Design-Pattern/Pizza.cpp at master · truongvanhuy2000/Design-Pattern (github.com)

TEMPLATE PATTERN

➢ Bài toán đặt ra

Bạn cần viết Menu cho 2 loại thức uống caffein là Coffee và trà như sau:

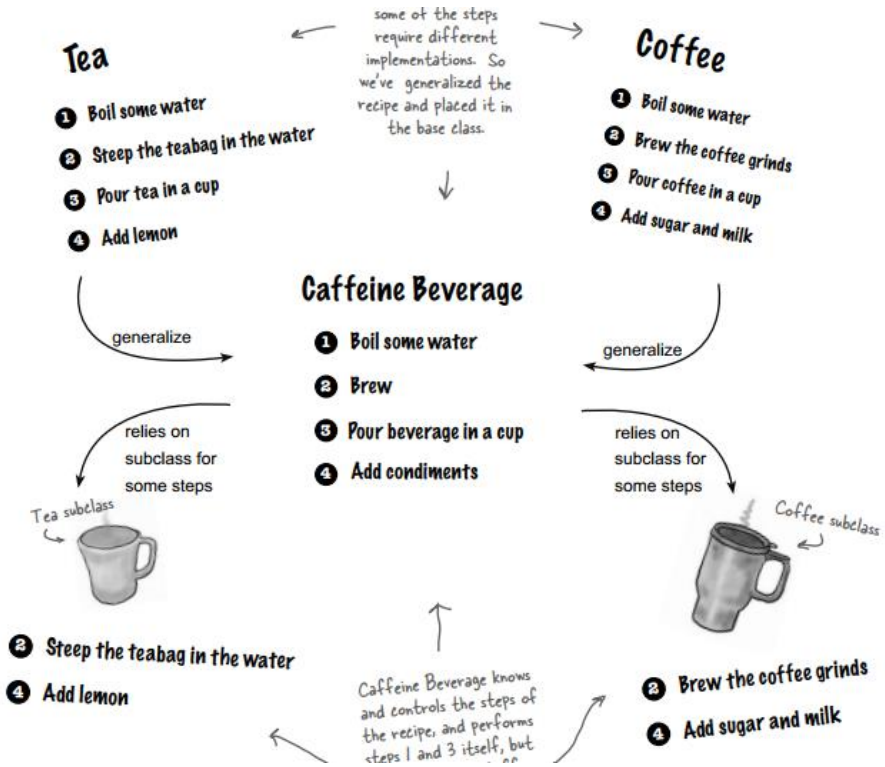
Nếu chỉ đơn giản là tạo 2 Class cho 2 loại thức uống thì sẽ thấy lặp code khá nhiều, nếu có nhiều loại thức uống caffein nữa thì vấn đề sẽ tệ hơn nhiều.



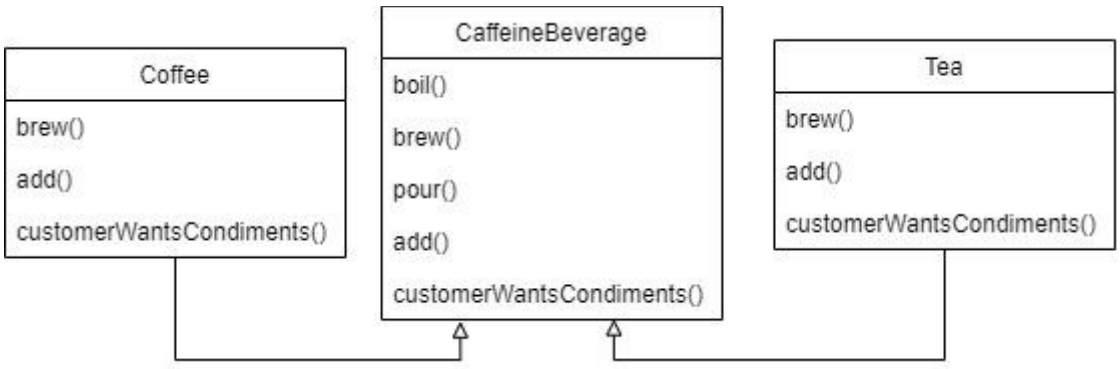
➢ Giới thiệu Template Pattern

Template Pattern xác định các bước của một thuật toán và cho phép các lớp con cung cấp việc triển khai một hoặc nhiều bước.

- Chúng ta sẽ khái quát các bước để tạo ra đồ uống caffeine để tạo ra một Abstract Class chung cho chúng
- Các class con sẽ kế thừa và áp dụng cách thức hoạt động riêng của chúng ở một vài bước



➢ Lưu đồ



➢ Code

Class chung cho các đồ uống caffeine

```
class CaffeineBeverage
{
    public:
```

```
        void prepareRecipe()
        {
            boil();
            brew();
            pour();
            if(customerWantsCondiments())
            {
                add();
            }
        }
        virtual void boil()
        {
            cout << "boiling water" << endl;
        }
        virtual void brew()
        {
            cout << "brewing thing" << endl;
        }
        virtual void pour()
        {
            cout << "pour in cup" << endl;
        }
        virtual void add()
        {
            cout << "add condiments" << endl;
        }
        virtual bool customerWantsCondiments()
        {
            return true;
        }
};
```

Class cho 1 loại thức uống Caffine

```
class Coffee : public CaffeineBeverage
{
    public:
        void brew()
        {
            cout << "Brew the coffee grinds" << endl;
        }
        void add()
        {
            cout << "Add sugar and milk" << endl;
        }
        bool customerWantsCondiments()
        {
            char x;
            cout << "want condiments or not? (Y/N)" << endl;
            cin >> x;
            if(x == 'y')
            {
                return true;
            }
            else
```

```
            {
                return false;
            }
        }
};
```

Hàm main()

```
int main()
{
    CaffeineBeverage* tea = new Tea();
    CaffeineBeverage* coffee = new Coffee();
    cout << "Prepare tea" << endl;
    tea->prepareRecipe();
    cout << "Prepare coffee" << endl;
    coffee->prepareRecipe();
    return 0;
}
```

- Kết quả chạy

```
Prepare tea
boiling water
Steep the teabag in the water
pour in cup
want condiments or not? (Y/N)
y
Add lemon
Prepare coffee
boiling water
Brew the coffee grinds
pour in cup
want condiments or not? (Y/N)
y
Add sugar and milk
```

Link code đầy đủ: [Design-Pattern/StarBuzz.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

ITERATOR PATTERN

➢ Bài toán đặt ra

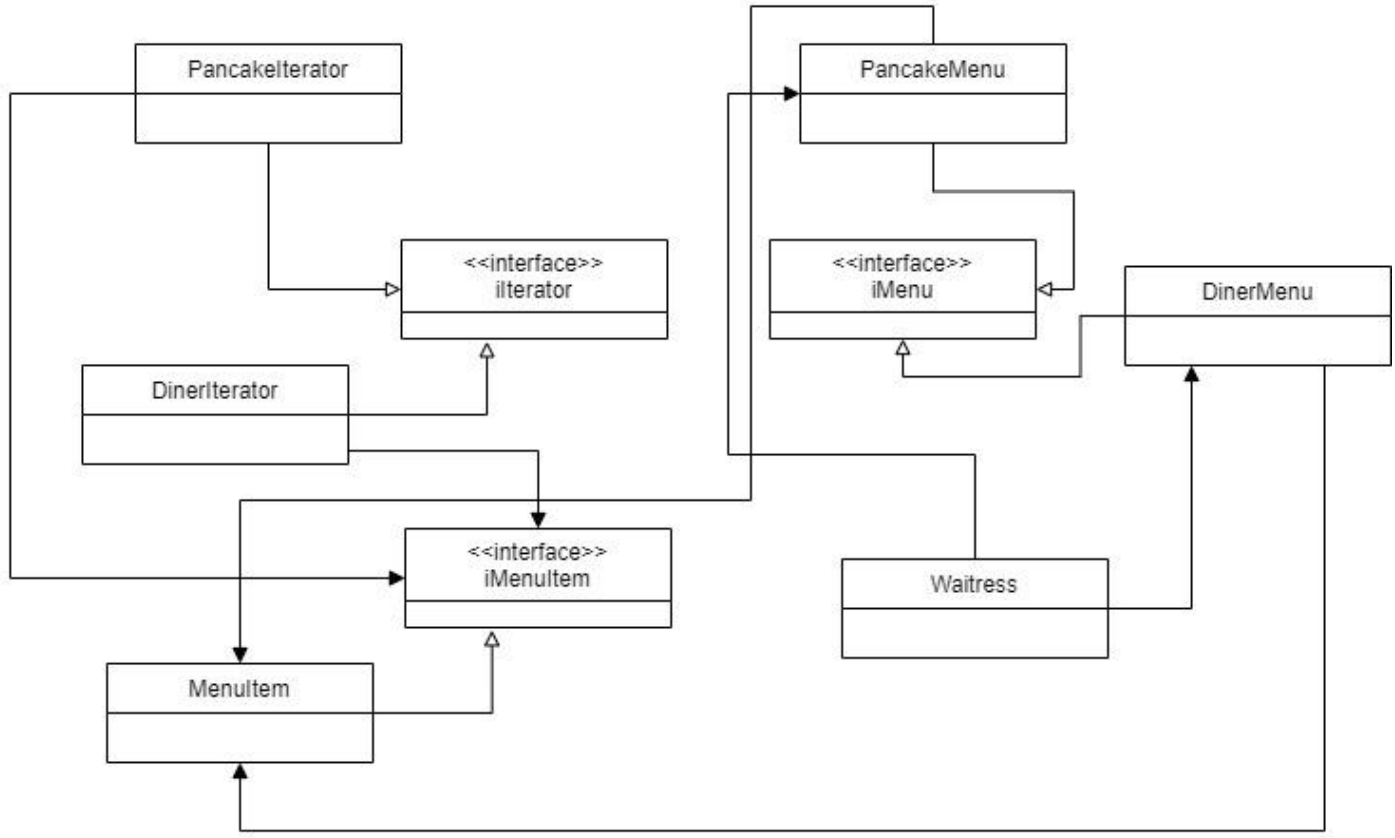
Tin nóng: Objectville Diner và Objectville Pancake House hợp nhất. Tin tốt đấy! Giờ đây, chúng ta có thể thưởng thức những bữa sáng pancake ngon lành của Pancake House và những bữa trưa của Diner ở cùng một nơi. Thế nhưng 2 đầu bếp lại sử dụng 2 phương thức khác nhau để lưu trữ menu, 1 người dùng vector, 1 người dùng array. Ít nhất thì 2 đầu bếp đồng ý sử dụng chung 1 menu Item.

➢ Giới thiệu Iterator Pattern

Iterator Pattern cung cấp một cách để truy cập tuần tự các phần tử của một đối tượng mà không để lộ tầng dưới của nó.

Yếu tố quan trọng khác đối với thiết kế của bạn là Iterator Pattern có trách nhiệm duyệt các phần tử và trao trách nhiệm đó cho Iterator object, không phải object tổng hợp. Điều này không chỉ giữ cho interface và triển khai cho object tổng hợp đơn giản hơn mà còn loại bỏ trách nhiệm duyệt phần tử khỏi object tổng hợp.

➢ Lưu đồ



➢ Code

Interface cho iMenuItem

```
class iMenuItem
{
    public:
        virtual string getName() = 0;
        virtual string getDescription() = 0;
        virtual bool isVegetarian() = 0;
        virtual double getPrice() = 0;
};
```

Interface cho iIterator

```
class iIterator
```

```
{
    public:
        virtual bool hasNext() = 0;
        virtual iMenuItem* next() = 0;
};
```

Interface cho iMenu

```
class iMenu
{
    public:
        virtual iIterator* createIterator() = 0;
};
```

Class PancakeIterator sẽ có nhiệm vụ duyệt các thành viên của Pancake vector

```
class PancakeIterator : public iIterator
{
    vector<iMenuItem*> mMenuItem;
    int mPosition = 0;
    public:
        PancakeIterator(vector<iMenuItem*> menuItem)
        {
            mMenuItem = menuItem;
        }
        bool hasNext()
        {
            if(mPosition >= mMenuItem.capacity())
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        iMenuItem* next()
        {
            iMenuItem* menItem = mMenuItem[mPosition];
            mPosition++;
            return menItem;
        }
};
```

Class DinerIterator sẽ có nhiệm vụ duyệt các thành viên của Diner Array

```
class DinerIterator : public iIterator
{
    iMenuItem* *mMenuItem;
    int mMaxItem;
    int mPosition = 0;
    public:
        DinerIterator(iMenuItem* *menuItem, int maxItem)
        {
            mMenuItem = menuItem;
            mMaxItem = maxItem;
        }
        bool hasNext()
        {
```

```
            if(mPosition >= mMaxItem)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        iMenuItem* next()
        {
            iMenuItem* menItem = *(mMenuItem + mPosition);
            mPosition++;
            return menItem;
        }
};
```

Ví dụ cho Menu

```
class DinerMenu : public iMenu
{
    private:
        int mMaxItem = 3;
        iMenuItem* mMenuItem[3];
        int mPosition = 0;;
    public:
        DinerMenu()
        {
            addItem("diner", "abcd", true, 2.99);
            addItem("diner", "bcde", false, 1.99);
            addItem("diner", "456", false, 0.99);
        }
        void addItem(string name, string description,bool vegeterian, double price)
        {
            iMenuItem* menuItem = new MenuItem(name, description, vegeterian, price);
            mMenuItem[mPosition] = menuItem;
            mPosition++;
        }
        iIterator* createIterator()
        {
            return new DinerIterator(mMenuItem, mMaxItem);
        }
};
```

Việc của class Waitress sẽ chỉ là nhận các phần tử được lấy ra từ Iterator sau đó in ra, không cần quan tâm dữ liệu được lấy ra như nào

```
class Waitress
{
    iMenu *mPancakeMenu;
    iMenu *mDinerMenu;
    void printMenu(iIterator *menuIterator)
    {
        while(menuIterator->hasNext())
        {
            iMenuItem* menuItem = menuIterator->next();
```

```
            cout << " Name: " << menuItem->getName() <<" Description: " << menuItem-
>getDescription() << " Price: " << menuItem->getPrice() << endl;
        }
    }
    public:
        Waitress(iMenu *PancakeMenu, iMenu *DinerMenu)
        {
            mPancakeMenu = PancakeMenu;
            mDinerMenu = DinerMenu;
        }
        void printMenu()
        {
            iIterator *pancakeIterator = mPancakeMenu->createIterator();
            iIterator *dinerIterator = mDinerMenu->createIterator();
            cout << "Pancake Menu" << endl;
            printMenu(pancakeIterator);
            cout << "Diner Menu" << endl;
            printMenu(dinerIterator);
        }
};
```

Hàm main()

```
int main()
{
    PancakeMenu pankeMenu1;
    DinerMenu dinerMenu1;
    Waitress waitress1(&pankeMenu1, &dinerMenu1);
    waitress1.printMenu();
}
```

- Kết quả chạy:

```
Pancake Menu
 Name: cake Description: abcd Price: 2.99
 Name: cake Description: bcde Price: 1.99
 Name: cake Description: 456 Price: 0.99
Diner Menu
 Name: diner Description: abcd Price: 2.99
 Name: diner Description: bcde Price: 1.99
 Name: diner Description: 456 Price: 0.99
```

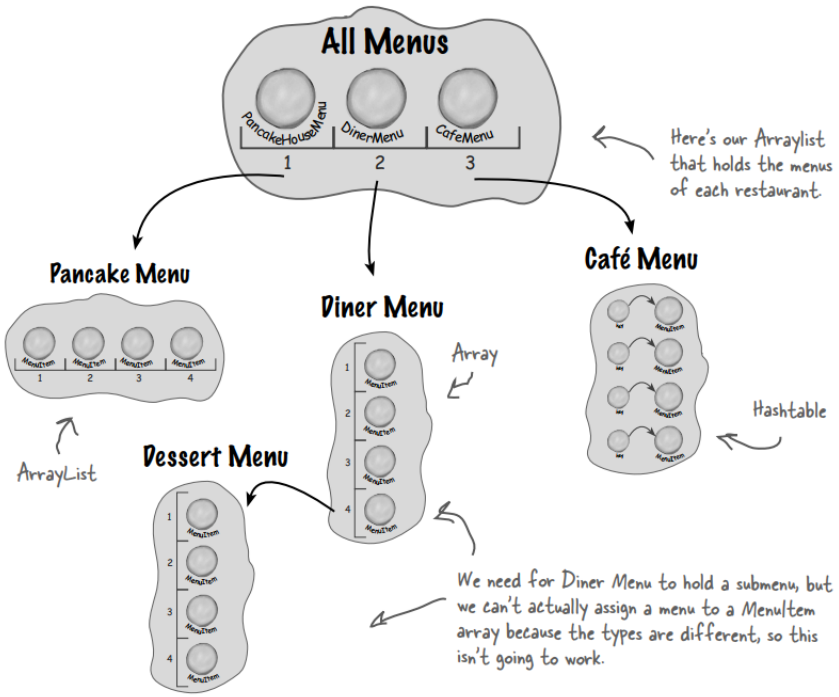
Link code đầy đủ: [Design-Pattern/Menu.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]

COMPOSITE PATTERN

➢ Bài toán đặt ra

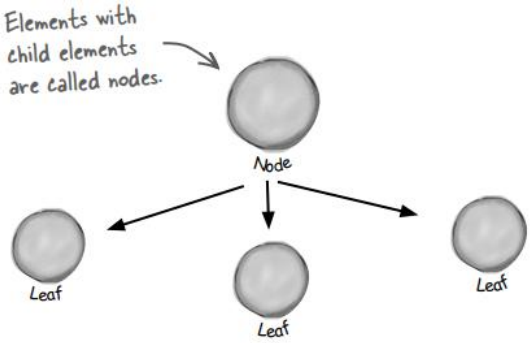
Giống với bài toán ở Iterator Pattern, thế nhưng giờ người ta muốn thêm Menu phụ ngay trong Menu chính. Thế là chúng ta phải đập đi xây lại cấu trúc menu đã xây dựng ở Iterator Pattern

- Chúng ta cần một loại cấu trúc hình cây có thể chứa các menu, menu con và các mục của menu
- Chúng ta cần đảm bảo duy trì một cách để duyệt các mục trong mỗi menu thuận tiện như những gì chúng ta đang làm với Iterator Pattern
- Chúng ta cần phải có khả năng duyệt các menu một cách linh hoạt hơn.

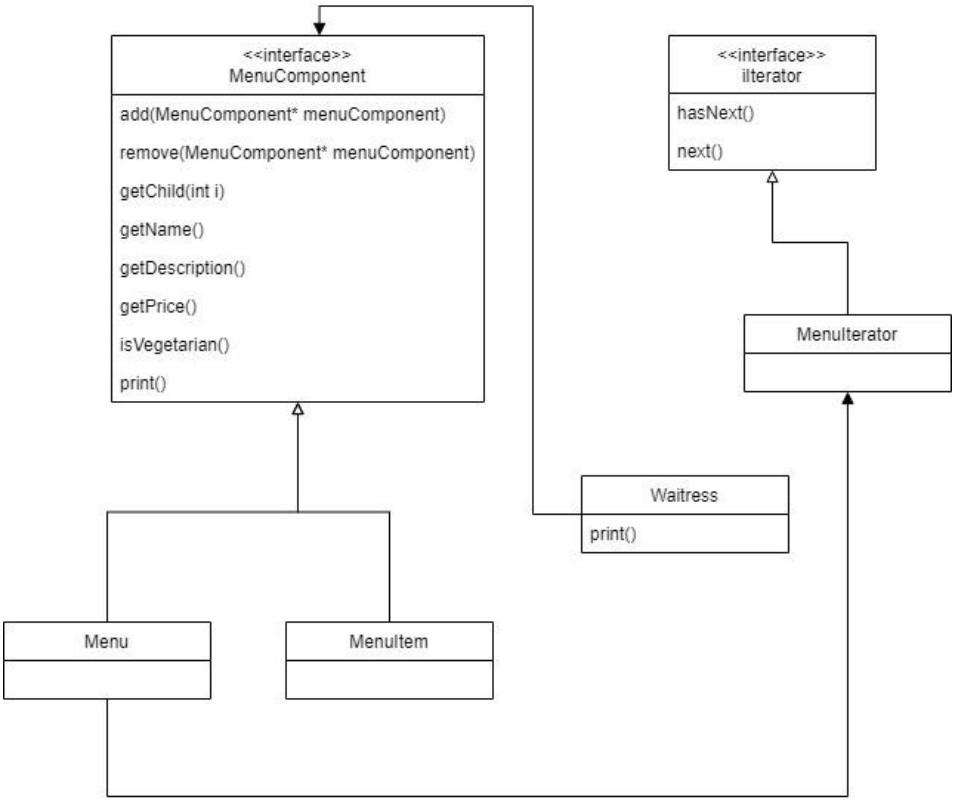


➢ Giới thiệu Composite Pattern

Composite Pattern cho phép bạn sắp xếp đối tượng thành cấu trúc dạng cây để biểu diễn sự phân cấp. Composite cho phép khách hàng xử lý các đối tượng riêng lẻ và bố cục của các đối tượng một cách thống nhất



➢ Lưu đồ



- Code

Interface chung cho các node

```
class MenuComponent
{
    public:
        virtual void add(MenuComponent* menuComponent){}
        virtual void remove(MenuComponent* menuComponent){}
        virtual MenuComponent* getChild(int i){return 0;}
        virtual string getName(){return 0;}
        virtual string getDescription(){return 0;}
        virtual double getPrice(){return 0;}
        virtual bool isVegetarian(){return 0;}
        virtual void print(){}
};
```

Interface cho Iterator, đúng vậy, chúng ta sẽ dùng lại Iterator Interface

```
class iIterator
{
    public:
        virtual bool hasNext() = 0;
        virtual MenuComponent* next() = 0;
};
```

Iterator cho MenuComponent

```
class MenuIterator : public iIterator
{
    vector<MenuComponent*> mMenu;
    int mPosition = 0;
    public:
        MenuIterator(vector<MenuComponent*> menu)
        {
            mMenu = menu;
        }
        bool hasNext()
        {
            if(mPosition >= mMenu.capacity())
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        MenuComponent* next()
        {
            MenuComponent* menu = mMenu[mPosition];
            mPosition++;
            return menu;
        }
};
```

Class cho các món ăn trong Menu

```
class MenuItem : public MenuComponent
```

```
{
    string mName;
    string mDescription;
    bool mVegetarian;
    double mPrice;
    public:
        MenuItem(string name, string description,bool vegeterian, double price)
        {
            mName = name;
            mDescription = description;
            mVegetarian = vegeterian;
            mPrice = price;
        }
        string getName()
        {
            return mName;
        }
        string getDescription()
        {
            return mDescription;
        }
        bool isVegetarian()
        {
            return mVegetarian;
        }
        double getPrice()
        {
            return mPrice;
        }
        void print()
        {
            cout << " Name: " << getName() <<" Description: " << getDescription() << "
Price: " << getPrice() << endl;
        }
};
```

Class cho các Menu lớn

```
class Menu : public MenuComponent
{
    vector<MenuComponent*> mMenuList;
    string mName;
    string mDescription;
    public:
        Menu(string name, string description)
        {
            mName = name;
            mDescription = description;
        }
        void add(MenuComponent* menuComponent)
        {
            mMenuList.push_back(menuComponent);
        }
```

```
        void remove(MenuComponent* menuComponent)
        {
            for (int i = 0; i < mMenuList.capacity(); i++)
            {
                if(menuComponent == mMenuList[i])
                {
                    mMenuList.erase(mMenuList.begin() + i);
                    break;
                }
            }
        }
        MenuComponent* getChild(int i)
        {
            return mMenuList[i];
        }
        string getName()
        {
            return mName;
        }
        string getDescription()
        {
            return mDescription;
        }
        void print()
        {
            cout << getName() << endl;
            cout << getDescription() << endl;
            cout << "---------------------" << endl;;
            iIterator* iterator = new MenuIterator(mMenuList);
            while(iterator->hasNext())
            {
                MenuComponent* menuComponent = iterator->next();
                menuComponent->print();
            }
        }
};
```

Class cho Waitress, giao diện của Waitress rất đơn giản, còn đơn giản hơn rất nhiều so với Iterator Pattern

```
class Waitress
{
    public:
        Waitress(){}
        void print(MenuComponent* allMenu)
        {
            allMenu->print();
        }
};
```

Hàm Main()

```
int main()
{
    MenuComponent* pancakeHouseMenu = new Menu("PANCAKE HOUSE MENU", "Breakfast");
    MenuComponent* dinerMenu = new Menu("DINER MENU", "Lunch");
    MenuComponent* cafeMenu = new Menu("CAFE MENU", "Dinner");
```

```
    MenuComponent* dessertMenu = new Menu("DESSERT MENU", "Dessert of course!");
    MenuComponent* allMenus = new Menu("ALL MENUS", "All menus combined");
    allMenus->add(pancakeHouseMenu);
    allMenus->add(dinerMenu);
    allMenus->add(cafeMenu);
    allMenus->add(dessertMenu);
    dinerMenu->add(new MenuItem("Pasta", "Spaghetti with Marinara Sauce, and a slice of sourdough bread", true, 3.89));
    dessertMenu->add(new MenuItem("Apple Pie", "Apple pie with a flakey crust, topped with vanilla icecream", true, 1.59));
    dessertMenu->add(new MenuItem("123456", "456789", true, 1.59));
    pancakeHouseMenu->add(dessertMenu);
    dinerMenu->add(dessertMenu);
    Waitress waitress;
    cout << "here your full Menu" << endl;
    waitress.print(allMenus);
    cout << "here your Diner Menu" << endl;
    waitress.print(dinerMenu);
    return 0;
}
```

Kết quả chạy:

```
here your full Menu
ALL MENUS
All menus combined
---------------------
PANCAKE HOUSE MENU
Breakfast
---------------------
DESSERT MENU
Dessert of course!
---------------------
 Name: Apple Pie Description: Apple pie with a flakey crust, topped with vanilla icecream Price: 1.59
 Name: 123456 Description: 456789 Price: 1.59
DINER MENU
Lunch
---------------------
 Name: Pasta Description: Spaghetti with Marinara Sauce, and a slice of sourdough bread Price: 3.89
DESSERT MENU
Dessert of course!
---------------------
 Name: Apple Pie Description: Apple pie with a flakey crust, topped with vanilla icecream Price: 1.59
 Name: 123456 Description: 456789 Price: 1.59
CAFE MENU
Dinner
---------------------
DESSERT MENU
Dessert of course!
---------------------
 Name: Apple Pie Description: Apple pie with a flakey crust, topped with vanilla icecream Price: 1.59
 Name: 123456 Description: 456789 Price: 1.59


 here your Diner Menu
DINER MENU
Lunch
---------------------
 Name: Pasta Description: Spaghetti with Marinara Sauce, and a slice of sourdough bread Price: 3.89
DESSERT MENU
Dessert of course!
---------------------
 Name: Apple Pie Description: Apple pie with a flakey crust, topped with vanilla icecream Price: 1.59
 Name: 123456 Description: 456789 Price: 1.59
```

Link code đầy đủ: [Design-Pattern/Tree.cpp at master · truongvanhuy2000/Design-Pattern (github.com)]