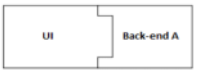
Structural Pattern

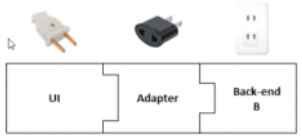
Tuesday, June 29, 2021 3:53 PM

Adapter Pattern

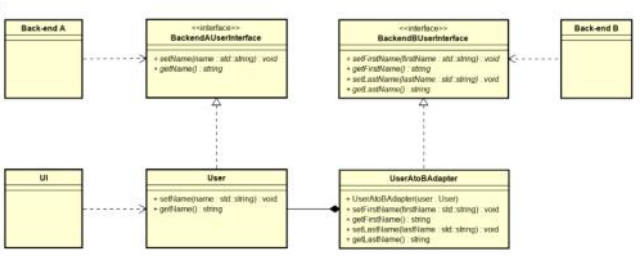
Ví dụ:

- Ứng dụng bao gồm 2 phần chính là UI (User Interface) và Back-end (chịu trách nhiệm xử lý về logic, đọc / ghi database, đưa dữ liệu lên cho UI hiển thị)
- Phần Back-end chúng ta đang sử dụng được phát triển bởi một công ty khác, tạm gọi là công ty A đi.
- công ty quyết định chúng ta phải chuyển sang dùng Back-end của công ty B. Và có một vấn đề xảy ra, đó là các Interfaces của Back-end A và Back-end B khác nhau (khác nhau về function name, về danh sách parameters, kiểu dữ liệu của paramaters) do đó nếu muốn dùng Back-end B thì chúng ta đang đứng trước nguy cơ phải sửa rất nhiều code
- => Chúng ta sẽ dùng Adapter Pattern để chuyển đổi interface của Back-end B sang interface của Back-end A, điều đó sẽ giúp cho UI có thể làm việc được với Back-end B mà không cần phải sửa code (hoặc sửa rất ít)





Class Diagram:



Ban đầu:

```
class BackendAUserInterface
{
public:
   virtual void setName(std::string name) = 0;
   virtual std::string getName() = 0;
};
```

User kế thừa BackendAUserInterface:

```
class User : public BackendAUserInterface
{
private:
   std::string mName;

public:
   void setName(std::string name)
   {
      mName = name;
   }

   std::string getName()
   {
      return mName;
   }
};
```

Main: ở UI khởi tạo và implement các phương thức của User

```
int main()
{
   // create User object
   User user;
   user.setName("Tuan Pham Minh");
  cout << "name: " << user.getName() << endl;
}
```

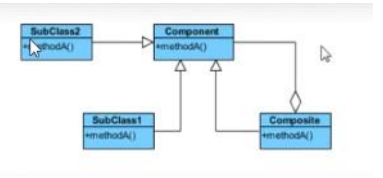
Bài toán: Có thêm Backend B và user muốn implement các phương thức của B thay A
=> thông thường sẽ khởi tạo User B kế thừa interface BE B sau đó những chỗ nào có User thay bằng User B

```
//Interface Backend B
class BackendBUserInterface
{
public:
   virtual void setFirstName(std::string firstName) = 0;
   virtual void setLastName(std::string lastName) = 0;
   virtual std::string getFirstName() = 0;
   virtual std::string getLastName() = 0;
};
```

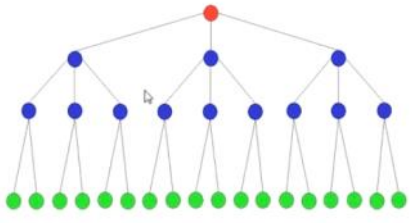
Composite Pattern

- Composite là một mẫu thiết kế cấu trúc cho phép sắp xếp các đối tượng thành một cấu trúc giống như cây và làm việc với nó như thể nó là một đối tượng đơn lẻ.
- Composite đã trở thành một giải pháp khá phổ biến cho hầu hết các vấn đề yêu cầu xây dựng cấu trúc cây. Tính năng tuyệt vời của Composite là khả năng chạy các phương thức đệ quy trên toàn bộ cấu trúc cây và tổng hợp kết quả.

Tổ hợp: Bài toán 1 ứng dụng Word tổ hợp của các chức năng Home, Insert, Page Layout,...
Trong các chức năng thì Insert là tổ hợp của các chức năng khác như table, picture, clicp art, ..
...
=> Composite là tổng hợp của các thằng nhỏ có quan hệ với nhau để tạo ra thằng lớn hơn
Ở UML này, các subclass, Composite là các lớp con kế thừa component
Composite là tổ hợp của các thằng component khác(dấu quan hệ hình thoi asociate)
Composite đây là Word, các thằng home, insert là các component



Nút đỏ là composite của tất cả. Còn lại dưới là các component. Các component lại là composite của các nút dưới nó nữa



Ví dụ:

//Class Component:

```
class Component {
protected:
  Component *parent_;
public:
  virtual ~Component() {}
  void SetParent(Component *parent) {
    this->parent_ = parent;
  }
  Component *GetParent() const {
    return this->parent_;
  }
  /**
   * Trong một số trường hợp, sẽ có lợi nếu xác định các hoạt động quản lý lớp
   * con ngay trong lớp Component. Bằng cách này, bạn sẽ không cần hiển thị
   * bất kỳ lớp component cụ thể nào với client, ngay cả trong quá trình tạo cây
   * đối tượng. Nhược điểm là các phương thức này sẽ rỗng đối với các nút leaf.
   */
  virtual void Add(Component *component) {}
  virtual void Remove(Component *component) {}
    virtual bool IsComposite() const {
    return false;
  }
  /**
   * Component có thể thực hiện một số hành vi mặc định hoặc để nó
   * cho các lớp cụ thể (bằng cách khai báo phương thức trừu tượng).
   */
  virtual std::string Operation() const = 0;
};
```

//Các subClass (lớp lá) kế thừa từ component:

```
class Leaf : public Component {
public:
  std::string Operation() const override {
    return "Leaf";
  }
};
```

//Class composite cũng kế thừa từ Component, ở đây implenment các phương thức ảo của component và tổng hợp các component thành 1 list, ở hàm operation, duyệt các component có sẵn rồi gọi đến các operation của các component từ ngoài vào lớp lá cuối cùng

```
class Composite : public Component {
protected:
  std::list&lt;Component *> children_;
```

```
public:
  virtual void setFirstName(std::string firstName) = 0;
  virtual void setLastName(std::string lastName) = 0;
  virtual std::string getFirstName() = 0;
  virtual std::string getLastName() = 0;
};
```

Solution:
Tạo 1 thằng adapter **UserAtoBAdapter** kế thừa BackendBUserInterface
Trong đó hàm khởi tạo truyền vào thằng User hiện có
Sau đó sử dụng những phương thức có sẵn của BackendA để chuyển sang BackendB

```
class UserAtoBAdapter : public BackendBUserInterface
{
private:
  User mUser;

public:
  UserAtoBAdapter(User user)
  {
    mUser = user;
    // split first name and last name
    unsigned int splitPostion = user.getName().find_first_of(" ");
    if (splitPostion != string::npos)
    {
      mFirstname = user.getName().substr(0, splitPostion + 1);
      mLastname = user.getName().substr(splitPostion + 1, user.getName().length() - mFirstname.length());
    }
  }

  void setFirstName(std::string firstName)
  {
    mFirstname = firstName;
  }

  void setLastName(std::string lastName)
  {
    mLastname = lastName;
  }
...
};
```

Ở hàm main chỗ nào cần sửa chỉ cần:

- UserAtoBAdapter adapter(user);
  Sau đó có thể dùng đối tượng adaper như User thay thế cho User A ở các nơi cần thiết

Tóm lại:

- Quay lại bài toán ban đầu thì như vậy sau khi tạo đối tượng adapter, chúng ta có thể truyền nó vào cho Back-end B. Back-end B chỉ cần biết là nó đang làm việc với một đối tượng của class đã implement interface mà nó đưa ra là *BackendBUserInterface* mà không hề biết đến sự tồn tại của *User* hay *BackendAUserInterface*. Bằng cách đó chúng ta sẽ không phải sửa quá nhiều code để chuyển từ Back-end A sang làm việc với Back-end B.

- **Và trong tương lai nếu có chuyển sang dùng Back-end C, Back-end D, … thì cũng chỉ cần tạo class adapter mới và thay thế adapter mà thôi.**
- **Backend A đã thực hiện xử lý 1 số việc cần thiết Backend B có thể tận dụng và chuyển sang dùng các phương thức từ B tránh tạo lại UserB từ đầu lúc đấy sẽ thêm và sửa rất nhiều code có thể BackendA đã làm hoặc đã xử lý**

Façade Pattern

Facade là **một đối tượng** và đối tượng này cung cấp một **interface** đơn giản để che giấu đi các xử lý phức tạp bên trong nó.
Giả sử chúng ta xây dựng một lớp thư viện vậy Facade có thể:

- Giúp cho một thư viện của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, **vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.**
- **Giúp cho các đoạn mã có sử dụng thư viện trở nên dễ đọc hơn**, cũng lí do như trên.
- Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.
- Đóng gói tập nhiều hàm API được thiết kế không tốt bằng một hàm API đơn có thiết kế tốt hơn.

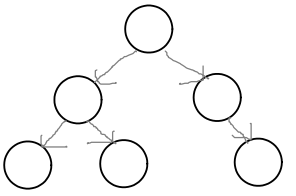
Đây chính là một ưu điểm của Facade, và dĩ nhiên cũng có câu trả lời cho vì sao lại nên sử dụng nó.

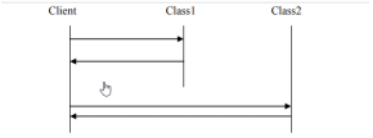
- Những gì bạn cần phải làm chỉ là thiết kế một Facade, và trong đó **phương thức facade sẽ xử lý các đoạn code dùng đi dùng lại**. Từ xu hướng quan điểm trên, chúng ta sẽ chỉ cần gọi **Facade để thực thi các hành động dựa trên các parameters được cung cấp**.
- Bây giờ nếu chúng ta cần bất kỳ thay đổi nào trong quá trình trên, công việc sẽ đơn giản hơn rất nhiều, chỉ cần thay đổi các xử lý trong phương thức facade của bạn và mọi thứ sẽ được đồng bộ thay vì thực hiện sự thay đổi ở những nơi sử dụng cả chuỗi các mã code đó.

```
class Composite : public Component {
protected:
  std::list&lt;Component *> children_;

public:
  void Add(Component *component) override {
    this->children_.push_back(component);
    component->SetParent(this);
  }
  /**
   * Hãy nhớ rằng phương thức này xóa con trỏ đến danh sách nhưng không
   * giải phóng bộ nhớ, bạn nên thực hiện theo cách thủ công hoặc tốt hơn
   * là sử dụng con trỏ thông minh smart_pointer.
   */
  void Remove(Component *component) override {
    children_.remove(component);
    component->SetParent(nullptr);
  }
  bool IsComposite() const override {
    return true;
  }
  /**
   * Composite thực thi logic chính của nó theo một cách cụ thể. Nó duyệt đệ
   * quy qua tất cả các con của nó, thu thập và tổng hợp kết quả của chúng.
   * Vì các con của composite truyền những lệnh gọi này cho con của chúng
   * và cứ thế, kết quả là toàn bộ cây đối tượng được duyệt qua.
   */
  std::string Operation() const override {
    std::string result;
    for (const Component *c : children_) {
      if (c == children_.back()) {
        result += c->Operation();
      } else {
        result += c->Operation() + "+";
      }
    }
    return "Branch(" + result + ")";
  }
};
```

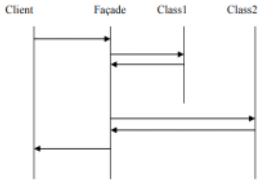
Ở hàm main:

```
int main() {
// client có thể hỗ trợ các thành phần lá, thành phần dưới cùng:
  Component *simple = new Leaf;
  std::cout &lt;&lt; "RESULT: " &lt;&lt; simple ->Operation();  => "RESULT: Leaf"
//Tạo 1 cây khúc phức tạp:
//Các tree các branch vẫn là các composite
  Component *tree = new Composite;
  Component *branch1 = new Composite;
  Component *branch2 = new Composite;

  // tạo các lá:
  Component *leaf_1 = new Leaf;
  Component *leaf_2 = new Leaf;
  Component *leaf_3 = new Leaf;

  // thêm các lá vào dưới các branch:
  branch1->Add(leaf_1);
  branch1->Add(leaf_2);
  branch2->Add(leaf_3);

//thêm các branch dưới tree:
  tree->Add(branch1);
  tree->Add(branch2);

=> Hình thành 1 cây khung
Implement phương thức từ tree:

std::cout &lt;&lt; "RESULT: " &lt;&lt; tree->Operation(); =>RESULT:
Branch(Branch(Leaf+Leaf)+Branch(Leaf))
Lần lượt gọi component từ các branch của tree, rồi các branch của tree tiếp tục call các Operation của các lớp lá dưới branch đó, đến lớp là thì dừng

// Hoặc có thể add thêm dưới composite tree, 1 nhánh leaf để hình thành cây khung khác, bla bla
Tree->Add(simple);
```





> If façade pattern is applied then the result will be like the following.



Ví dụ:

```
class Subsystem1 {
public:
 std::string Operation1() const {
   return "Subsystem1: Ready!\n";
 }
 // ...
 std::string OperationN() const {
   return "Subsystem1: Go!\n";
 }
};
/**
* Một số facade có thể hoạt động với nhiều hệ thống con cùng một lúc.
*/
class Subsystem2 {
public:
 std::string Operation1() const {
   return "Subsystem2: Get ready!\n";
 }
 // ...
 std::string OperationZ() const {
   return "Subsystem2: Fire!\n";
 }
};
```

// Khởi tạo Facade

```
class Facade {
protected:
 Subsystem1 *subsystem1_;
 Subsystem2 *subsystem2_;
 /**
  * Tùy thuộc vào nhu cầu của ứng dụng, bạn có thể cung cấp cho Facade
  * các đối tượng hệ thống con hiện có hoặc buộc Facade tự tạo chúng.
  */
public:
 /**
  * Trong trường hợp này, chúng ta sẽ ủy quyền quyền sở hữu bộ nhớ cho Facade
  */
 Facade(
     Subsystem1 *subsystem1 = nullptr,
     Subsystem2 *subsystem2 = nullptr) {
   this->subsystem1_ = subsystem1 ?: new Subsystem1;
   this->subsystem2_ = subsystem2 ?: new Subsystem2;
 }
 ~Facade() {
   delete subsystem1_;
   delete subsystem2_;
 }
 /**
  * Các phương thức của Facade là những lối tắt thuận tiện cho chức năng
  * phức tạp của các hệ thống con. Tuy nhiên, client chỉ nhận được một
  * phần nhỏ khả năng của hệ thống con.
  */
 std::string Operation() {
   std::string result = "Facade initializes subsystems:\n";
   result += this->subsystem1_->Operation1();
   result += this->subsystem2_->Operation1();
   result += "Facade orders subsystems to perform the action:\n";
   result += this->subsystem1_->OperationN();
   result += this->subsystem2_->OperationZ();
   return result;
 }
};
```

// ở hàm main:

```
int main() {
 Subsystem1 *subsystem1 = new Subsystem1;
 Subsystem2 *subsystem2 = new Subsystem2;
 Facade *facade = new Facade(subsystem1, subsystem2);

 std::cout &lt;&lt; facade->Operation();
 delete facade;

 return 0;
}
```

// Façade khác với việc tạo 1 hàm chung, trong hàm chung đó thực hiện các việc lặp đi lặp lại chỗ nào???
// Vì sao phải dùng Facade

Ví dụ từ wiki:

```
/* Complex parts */
class CPU {
   public void freeze() { ... }
   public void jump(long position) { ... }
```

Ví dụ từ wiki:

```
/* Complex parts */
class CPU {
   public void freeze() { ... }
   public void jump(long position) { ... }
   public void execute() { ... }
}
class Memory {
   public void load(long position, byte[] data) { ... }
}
class HardDrive {
   public byte[] read(long lba, int size) { ... }
}
/* Facade */
class ComputerFacade {
   private CPU processor;
   private Memory ram;
   private HardDrive hd;
public ComputerFacade() {
      this.processor = new CPU();
      this.ram = new Memory();
      this.hd = new HardDrive();
   }
public void start() {
      processor.freeze();
      ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
      processor.jump(BOOT_ADDRESS);
      processor.execute();
   }
}
/* Client */ Tính đóng gói, code đơn giản hơn thay vì khởi tạo lần lượt các đối tượng CPU, Memory, HardDrive và lần lượt thực hiện  các phương thức của từng thằng, bây giờ chỉ cần gọi start()
class You {
   public static void main(String[] args) {
      ComputerFacade computer = new ComputerFacade();
      computer.start();
   }
}
```