2023年春季学期

软件设计实践

面向可维护性的设计

谢 涛　　马 郓

本讲概览

- 软件维护
  - 提高可维护性的设计方法
- 面向可维护性的设计模式
  - 简单工厂模式、工厂方法模式、抽象工厂模式
  - 代理模式
  - 观察者模式、访问者模式、状态模式

软件维护

- 软件在投入使用之后，就进入维护阶段
  - 软件维护阶段是软件生命期中时间最长、花费精力和财力最多的阶段
  - 在软件运行/维护阶段对软件产品所进行的修改就是维护
- 4种类型的维护
  - 改正性维护（21%）
    - 在软件测试过程中，没有发现的错误，带到维护阶段，这些隐含的错误在某些特定的环境下会暴露出来，需要修复这些缺陷
  - 适应性维护（25%）
    - 随着计算机的发展，计算机硬件和软件环境都在不断地发生变化，为使软件适应这种变化需要修改软件
  - 完善性维护（50%）
    - 在软件使用过程中，用户往往会对软件提出新的功能要求与性能要求，为满足这些新的要求需要修改软件
  - 预防性维护（4%）
    - 为提高软件的可维护性和可靠性，为以后进一步改进软件奠定良好基础也需要修改软件

软件维护

- 软件维护不仅仅是运维工程师的工作
- 在设计与开发阶段就要考虑将来的可维护性
  - Maintainability 可维护性
  - Extensibility 可扩展性
  - Flexibility 灵活性
  - Adaptability 可适应性
  - Manageability 可管理性
- 提高可维护性的设计方法
  - 模块化编程
  - 遵循设计原则
  - 采用良好的设计模式

本讲概览

- 软件维护
  - 提高可维护性的设计方法
- **面向可维护性的设计模式**
  - **简单工厂模式、工厂方法模式、抽象工厂模式**
  - **代理模式**
  - **观察者模式、访问者模式、状态模式**

简单工厂模式

❑思考：

➢ 下面创建日志记录器对象的写法有什么潜在问题？

```
class Logger {
public:
  virtual void log(const string& msg) = 0;
};
```

```
// 控制台日志记录器
class ConsoleLogger : public Logger {
public:
  void log(const string& msg) {
    cout << "Console Logger: " << msg << endl;
  }
};
```

```
// 文件日志记录器
class FileLogger : public Logger {
public:
  void log(const string& msg) {
    cout << "File Logger: " << msg << endl;
    // 将日志写入文件
  }
};
```

```
// 数据库日志记录器
class DatabaseLogger : public Logger {
public:
  void log(const string& msg) {
    cout << "Database Logger: " << msg << endl;
    // 将日志写入数据库
  }
};
```

```
int main() {
    Logger* consoleLogger = new ConsoleLogger();
    consoleLogger->log("This is a console log.");

    Logger* fileLogger = new FileLogger();
    fileLogger->log("This is a file log.");

    Logger* databaseLogger = new DatabaseLogger();
    databaseLogger->log("This is a database log.");

    delete consoleLogger;
    delete fileLogger;
    delete databaseLogger;

    return 0;
}
```

潜在问题：

- 创建对象之前必须清楚所要创建对象的类信息，**客户端代码与具体的类紧耦合**
- 如果对象的创建需要一系列步骤，不仅仅只是new，则**创建逻辑需多次编写**，如果发生改变时需多处修改

简单工厂模式

- 解决办法：建立一个“工厂”类，用于创建不同的“产品”（即不同的子类）

```
class LoggerFactory {
public:
    static Logger* createLogger(const string& loggerType) {
        Logger* logger = nullptr;
        if (loggerType == "console") {
            logger = new ConsoleLogger();
        }
        else if (loggerType == "file") {
            logger = new FileLogger();
        }
        else if (loggerType == "database") {
            logger = new DatabaseLogger();
        }
        else {
            cout << "无法创建该类型的日志记录器" << endl;
        }
        return logger;
    }
};
```

简单工厂模式

- 解决办法：建立一个“工厂”类，用于创建不同的“产品”（即不同子类的对象）
  - 使用时只需通过工厂创建具体子类的对象

```
int main() {
    Logger* consoleLogger = LoggerFactory::createLogger("console");
    consoleLogger->log("This is a console log.");

    Logger* fileLogger = LoggerFactory::createLogger("file");
    fileLogger->log("This is a file log.");

    Logger* databaseLogger = LoggerFactory::createLogger("database");
    databaseLogger->log("This is a database log.");

    delete consoleLogger;
    delete fileLogger;
    delete databaseLogger;

    return 0;
}
```

有什么缺点？

简单工厂模式

- 优点：
  - 工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖
- 缺点：
  - 工厂类集中了所有实例的创建逻辑，它所能创建的类只能是事先考虑到的；如果需要添加新的类，就需要修改工厂类
  - 违反了“开放封闭原则”
- 如何优化？

工厂方法模式

- 优化办法：让每个类都有自己的工厂；当增加新的类时，只需增加新的工厂即可

```
// 抽象日志记录器工厂
class LoggerFactory {
public:
    virtual Logger* createLogger() = 0;
};
// 控制台日志记录器工厂
class ConsoleLoggerFactory : public LoggerFactory {
public:
    Logger* createLogger() {
        return new ConsoleLogger();
    }
};
// 文件日志记录器工厂
class FileLoggerFactory : public LoggerFactory {
public:
    Logger* createLogger() {
        return new FileLogger();
    }
};
// 数据库日志记录器工厂
class DatabaseLoggerFactory : public LoggerFactory {
public:
    Logger* createLogger() {
        return new DatabaseLogger();
    }
};
```

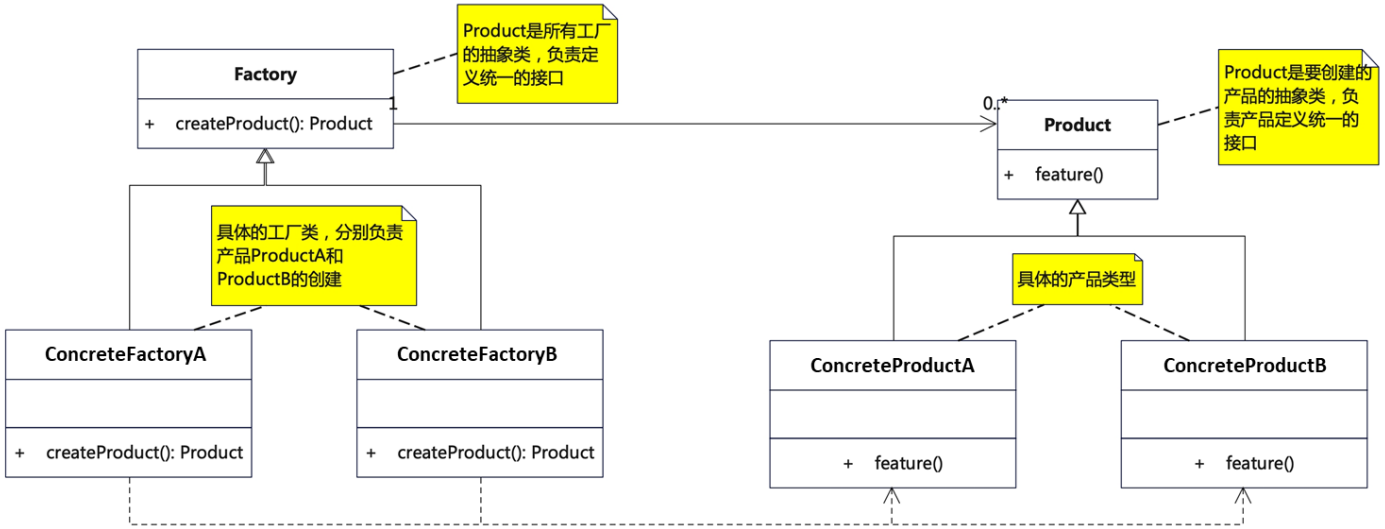
工厂方法模式

- 优化办法：让每个类都有自己的工厂；当增加新的类时，只需增加新的工厂即可
  - 使用时先创建工厂，再从工厂得到对象

```
int main() {
    // 使用控制台日志记录器工厂创建控制台日志记录器
    LoggerFactory* consoleFactory = new ConsoleLoggerFactory();
    Logger* consoleLogger = consoleFactory->createLogger();
    consoleLogger->log("This is a console log.");

    // 使用文件日志记录器工厂创建文件日志记录器
    LoggerFactory* fileFactory = new FileLoggerFactory();
    Logger* fileLogger = fileFactory->createLogger();
    fileLogger->log("This is a file log.");

    // 使用数据库日志记录器工厂创建数据库日志记录器
    LoggerFactory* databaseFactory = new DatabaseLoggerFactory();
    Logger* databaseLogger = databaseFactory->createLogger();
    databaseLogger->log("This is a database log.");

    // 省略delete

    return 0;
}
```

工厂方法模式

- 工厂方法（Factory Method）模式又称为工厂模式，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象
  - 将产品类的实例化操作延迟到工厂子类中完成



- **Product：** 定义工厂方法所创建对象的接口
- **ConcreteProduct：** 实现Product接口
- **Factory：** 声明工厂方法createProduct()，返回一个Product类型对象
- **ConcreteFactory：** 实现工厂方法，返回对应ConcreteProduct的对象

工厂方法模式

- 优点：
  - 工厂方法模式通过引入工厂等级结构，解决了简单工厂模式中工厂类职责太重的问题
    - 注：GoF的23种设计模式中并不包括简单工厂模式
- 缺点：
  - 由于每个工厂只生产一类产品，可能会导致系统中存在大量的工厂类，势必会增加系统的开销
- 此外，软件系统中经常面临“一系列相互依赖对象”的创建工作，由于需求变化，这“一系列相互依赖的对象”也要改变，如何应对这种变化呢？
  - 可以将这些对象一个个通过工厂方法模式来创建。但是，既然是一系列相互依赖的对象，它们是有联系的，每个对象都这样解决，如何保证它们的联系？

抽象工厂模式

- 思考：创建Windows和Mac的对话框和按钮类的对象

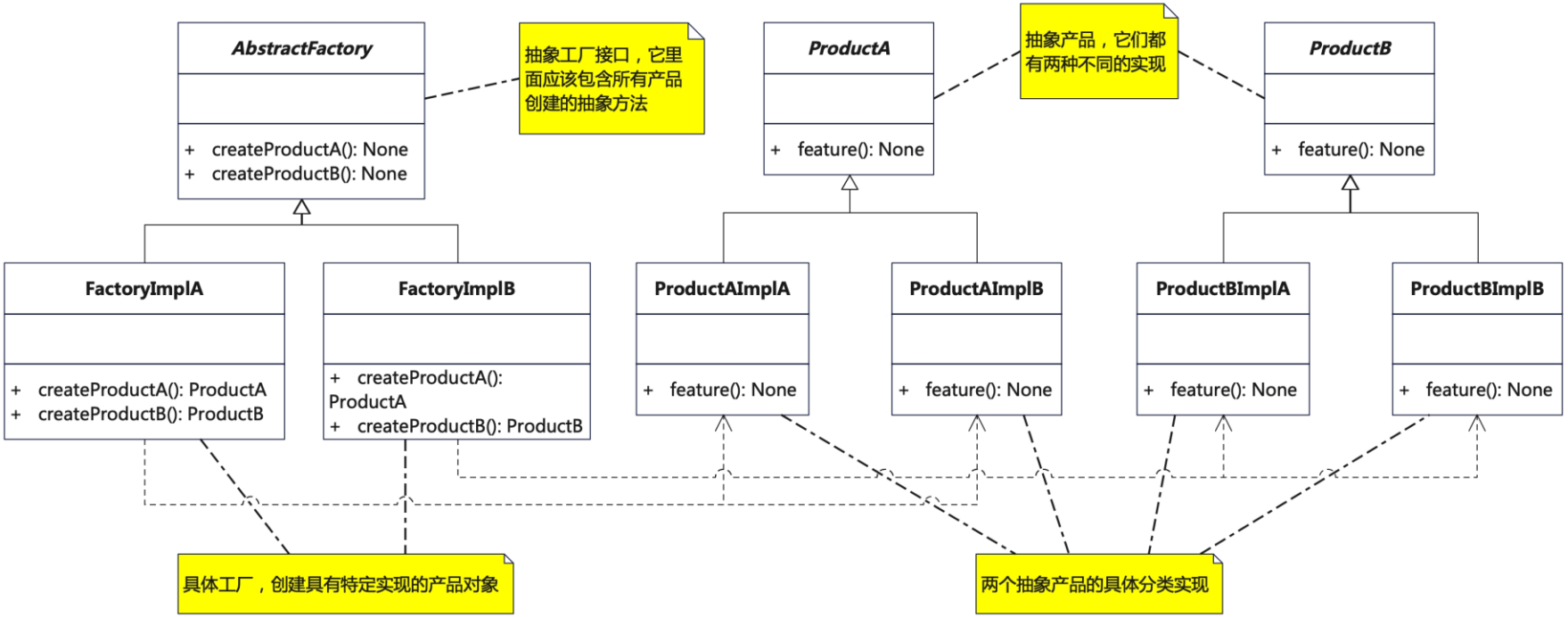
```
class Dialog {
public:
    virtual void render() = 0;
};

class Button {
public:
    virtual void onClick() = 0;
};
```

```
class WindowsDialog : public Dialog {
public:
    void render() {
        cout << "Rendering Windows Dialog" << endl;
    }
};
class WindowsButton : public Button {
public:
    void onClick() override {
        cout << "Windows Button Clicked" << endl;
    }
};
```

```
class MacDialog : public Dialog {
public:
    void render() {
        cout << "Rendering Mac Dialog" << endl;
    }
};
class MacButton : public Button {
public:
    void onClick() override {
        cout << "Mac Button Clicked" << endl;
    }
};
```

抽象工厂模式

- 抽象工厂（Abstract Factory）模式提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类
  - 适用于有多个系列且每个系列有相同子分类的产品



抽象工厂模式

- 例：创建Windows和Mac的对话框和按钮类的对象

```
class GUIFactory {
public:
    virtual Dialog* createDialog() = 0;
    virtual Button* createButton() = 0;
};

class WindowsFactory : public GUIFactory {
public:
    Dialog* createDialog() override {
        return new WindowsDialog();
    }
    Button* createButton() override {
        return new WindowsButton();
    }
};

class MacFactory : public GUIFactory {
public:
    Dialog* createDialog() override {
        return new MacDialog();
    }
    Button* createButton() override {
        return new MacButton();
    }
};
```

抽象工厂模式

- 例：创建Windows和Mac的对话框和按钮类的对象

```
int main() {
    // Create Windows GUI
    GUIFactory* windowsFactory = new WindowsFactory();
    Dialog* windowsDialog = windowsFactory->createDialog();
    Button* windowsButton = windowsFactory->createButton();
    windowsDialog->render();
    windowsButton->onClick();

    // Create Mac GUI
    GUIFactory* macFactory = new MacFactory();
    Dialog* macDialog = macFactory->createDialog();
    Button* macButton = macFactory->createButton();
    macDialog->render();
    macButton->onClick();
}
```

抽象工厂模式

- 优点：
  - 解决了具有二级分类的产品的创建
  - 易于交换产品系列，抽象工厂模式只需要改变具体工厂即可使用不同的产品配置
  - 让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户代码中
- 缺点：
  - 如果产品的分类超过二级，如三级甚至更多级，抽象工厂模式将会变得非常臃肿
  - 不能解决产品有多种分类、多种组合的问题

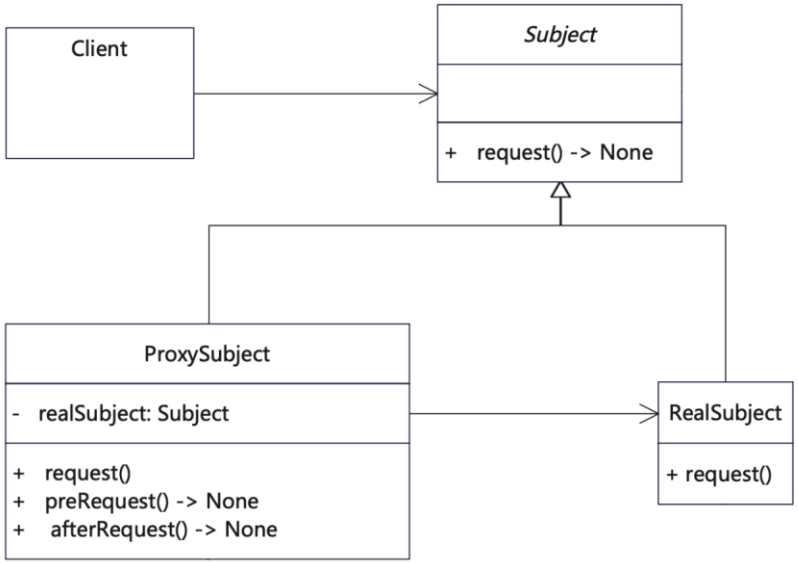
思考：

- 图像加载和显示类，如果图像很大、每次加载耗时很长，如何优化？

```
class RealImage {
private:
    string filename;
public:
    RealImage(const string& filename) : filename(filename) {
        // 加载图像的耗时操作
        loadFromDisk();
    }
    void display() {
        cout << "Displaying image: " << filename << endl;
    }
    void loadFromDisk() {
        cout << "Loading image: " << filename << endl;
        // 执行图像加载逻辑
    }
};
```

代理模式

- 代理（Proxy）模式为其他对象提供一种代理以控制对这个对象的访问
  - 虚代理：在需要创建开销很大对象时缓存对象信息
  - 远程代理：为一个对象在不同的地址空间提供局部代表
  - 保护代理：用来控制真实对象访问时的权限
  - 智能引用：当调用真实的对象时，代理处理另外一些事



Subject
- 定义RealSubject和Proxy的共用接口，在任何使用RealSubject的地方都可以使用Proxy

Proxy
- 保存一个引用使得代理可以访问实体
- 提供一个与Subject的接口相同的接口，这样代理就可以用来替代实体
- 控制对实体的存取，并可能负责创建和删除它
- 其他功能依赖于代理的类型

RealSubject
- 定义Proxy所代表的实体

❑ 例：大图像的加载

```
class Image {
public:
    virtual void display() = 0;
};

class RealImage : public Image {
private:
    string filename;
public:
    RealImage(const string&
filename) : filename(filename) {
        // 加载图像的耗时操作
        loadFromDisk();
    }
    void display() {
        cout << "Displaying image:
" << filename << endl;
    }
    void loadFromDisk() {
        cout << "Loading image: "
<< filename << endl;
        // 执行图像加载逻辑
    }
};
```

```
// 图像代理类
class ImageProxy : public Image {
private:
    string filename;
    RealImage* realImage;

public:
    ImageProxy(const string& filename) :
filename(filename), realImage(nullptr) {}

    void display() override {
        if (realImage == nullptr) {
            realImage = new RealImage(filename);
        }
        realImage->display();
    }
};
```

```
Image* image = new ImageProxy("image.jpg");
// 第一次显示图像，代理会加载实际图像
image->display();
// 第二次显示图像，代理不再加载实际图像
image->display();
```

❑ 例：利用代理对敏感数据增加权限控制

```
class SensitiveData {
public:
    virtual string readData() = 0;
};


class RealSensitiveData : public
SensitiveData {
    string data;
public:
    RealSensitiveData(const string&
data) : data(data) {}

    string readData() override {
        return data;
    }
};
```

```
class AccessControlProxy : public SensitiveData {
private:
    RealSensitiveData* realData;
    bool hasAccess;
public:
    AccessControlProxy(const string& data, bool
hasAccess)
        : realData(new RealSensitiveData(data)),
hasAccess(hasAccess) {}

    string readData() {
        if (hasAccess)
            return realData->readData();
        else
            return "Access Denied";
    }
};
```

```
// 创建访问控制代理对象
SensitiveData* data = new AccessControlProxy("Sensitive Information", false);
// 尝试读取数据
cout << "Data: " << data->readData() << endl;
```

代理模式

- 优点：
  - 代理模式能够协调调用者和被调用者，在一定程度上降低系统的耦合度
  - 可以灵活地隐藏被代理对象的部分功能和服务，也可以增加额外的功能和服务
- 缺点：
  - 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢
  - 实现代理模式需要额外的工作，有些代理模式的实现非常复杂
- 和适配器模式的区别：
  - 适配器模式主要改变所考虑对象的接口以消除不兼容
  - 代理模式目的是隔离对复杂对象的访问，不能改变所代理类的接口

观察者模式

❑思考：

- 下面的绘制数据图表的代码在可扩展性方面有什么问题？

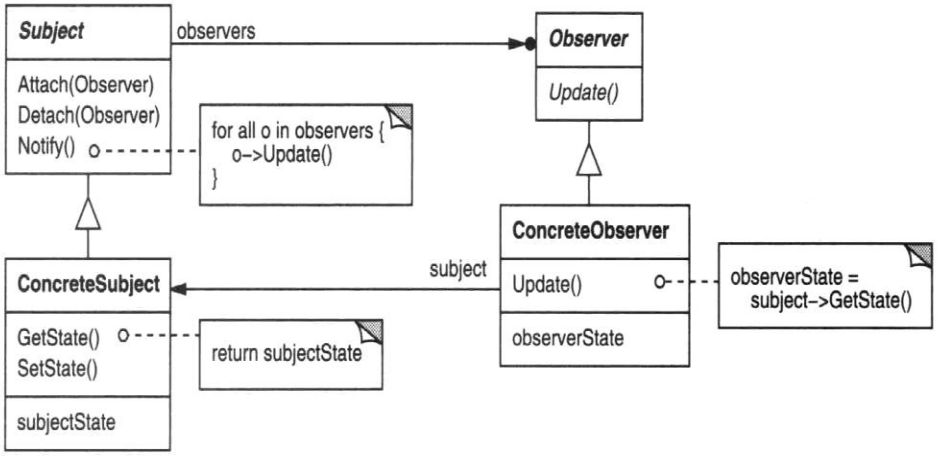
```
class DataSubject {
private:
    vector<int> data;
    Table to;
    Bar bo;
    Percentage po;

public:
  void setData(const
vector<int>& newData) {
    data = newData;
    draw();
  }

  void draw() {
    to.drawTable(data);
    bo.drawBar(data);
    po.drawPer(data);
  }
};
```

```
class Table {
public:
  void drawTable(const
vector<int>& data) {
    cout << "Table:" << endl;
    //根据data画表
  }
};
class Bar {
public:
  void drawBar(const
vector<int>& data) {
    cout << "Bar:" << endl;
    //根据data画柱状图
  }
};
class Percentage {
public:
  void drawPer(const
std::vector<int>& data) {
    cout << "Pstg:" << endl;
    //根据data画百分比图
  }
};
```

```
int main() {
  DataSubject dataSubject;

  // 设置初始数据
  vector<int> data = { 5, 2,
7, 3, 8, 4 };
  dataSubject.setData(data);

  // 修改数据
  data.push_back(6);
  dataSubject.setData(data);

  return 0;
}
```

观察者模式

- 问题：
  - 将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象间的一致性
- 观察者（Observer）模式定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新



- **Subject**：抽象的主体，即被观察的对象
- **Observer**：抽象的观察者
- **ConcreteSubject**：具体被观察对象
- **ConcreteObserver**：具体的观察者

注意：在观察者模式中，Subject通过Attach和Detach方法添加或删除所关联的观察者，并通过Notify进行更新，让每个观察者观察到最新的状态

观察者模式

❑例：绘制图表

```
class Observer {
public:
    virtual void update(const vector<int>& data) = 0;
};
```

```
class DataSubject {
  vector<int> data;
  vector<Observer*> observers;
public:
  void attach(Observer* observer) {
    observers.push_back(observer);
  }
  void detach(Observer* observer) {
    observers.erase(remove(observers.begin(),
observers.end(), observer), observers.end());
  }
  void setData(const vector<int>& newData) {
    data = newData;
    notifyObservers();
  }
private:
  void notifyObservers() {
    for (Observer* observer : observers) {
      observer->update(data);
    }
  }
};
```

```
// 表格观察者
class TableObserver : public Observer {
public:
  void update(const vector<int>& data) {
    cout << "Table:" << endl;
    //根据data画表
  }
};
// 柱状图观察者
class BarObserver : public Observer {
public:
  void update(const vector<int>& data) {
    cout << "Bar:" << endl;
    //根据data画柱状图
  }
};
// 百分比观察者
class PerObserver : public Observer {
public:
  void update(const vector<int>& data) {
    cout << “Pstg:" << endl;
    //根据data画百分比图
  }
};
```

例：绘制图表

```
int main() {
    DataSubject dataSubject;
    TableObserver tableObserver;
    BarObserver barChartObserver;
    PerObserver percentageObserver;
    // 注册观察者
    dataSubject.attach(&tableObserver);
    dataSubject.attach(&barChartObserver);
    dataSubject.attach(&percentageObserver);
    // 设置初始数据
    vector<int> data = { 5, 2, 7, 3, 8, 4 };
    dataSubject.setData(data);
    // 修改数据
    data.push_back(6);
    dataSubject.setData(data);
    // 取消观察者注册
    dataSubject.detach(&tableObserver);
    dataSubject.detach(&barChartObserver);
    dataSubject.detach(&percentageObserver);
}
```

观察者模式

应用场景

- 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变
- 对象仅需要将自己的更新通知给其他对象而不需要知道其他对象细节

优点

- Subject和Observer之间是松耦合的，可以各自独立改变
- Subject在发送广播通知时，无须指定具体的Observer，Observer可以自己决定是否要订阅Subject的通知
- 高内聚、低偶合

缺陷

- 松偶合导致代码关系不明显，有时可能难以理解
- 如果一个Subject被大量Observer订阅的话，在广播通知的时候可能会有效率问题

访问者模式

思考：

- 如何给Employees扩展新功能“增加假期”？

```
class Employee {
public:
    string name;
    double income;
    int vacationDays;

    Employee(string name, double income,
int vacationDays) {
        this->name = name;
        this->income = income;
        this->vacationDays =
vacationDays;
    }
};
```

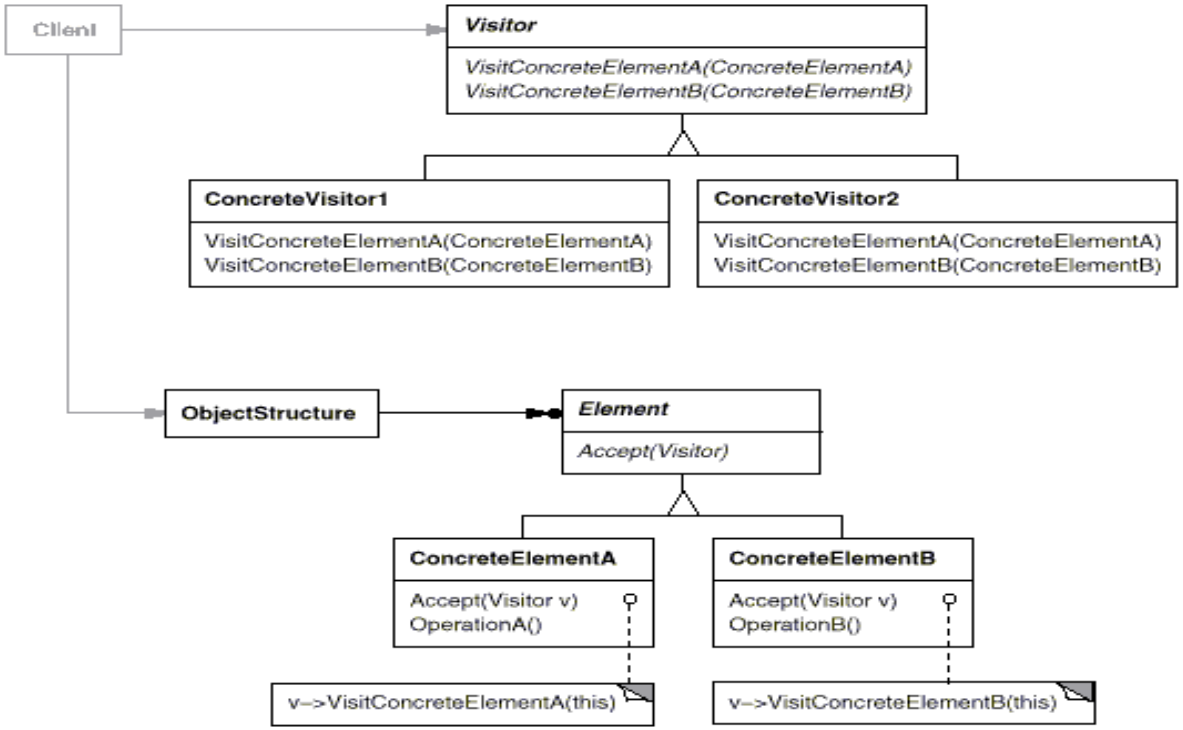
```
int main(int argc, char **argv) {
  Employees *e = new Employees();
  e->attach(new Employee("Tom", 25000.0, 14));
  e->attach(new Employee("Thomas", 35000.0, 16));
  e->attach(new Employee("Roy", 45000.0, 21));
  e->addIncome();
  return 0;
}
```

```
class Employees {
private:
    list<Employee *> employees;
public:
    void attach(Employee *employee) {
        employees.push_back(employee);
    }
    void detach(Employee *employee) {
        employees.remove(employee);
    }
    void addIncome() {
        for (auto employee:employees) {
            employee->income *= 1.10;
            cout << employee->name << "'s
new income: " << employee->income << endl;
        }
    }
};
```

访问者模式

- 问题：
  - 如何扩展一个现有的类层次结构来实现新行为？
    - 一般的方法是给类添加新的成员函数
    - 但是万一新行为和现有对象模型不兼容怎么办？
  - 类层次结构设计人员可能无法预知以后开发过程中将会需要哪些功能
    - 如果已有的类层次结构不允许修改代码，怎么能扩展行为呢？
- 访问者（Visitor）模式作用于某个对象群中各个对象的操作，可在不改变这些对象本身的情况下，定义作用于这些对象的新操作

访问者模式



- **Visitor：** 为对象结构中的具体元素提供一个访问操作接口。该操作接口的名字和参数标识了要访问的具体元素角色。访问者可以通过该元素角色的特定接口直接访问它
- **ConcreteVisitor：** 实现Visitor接口的操作
- **Element：** 该接口定义一个accept操作，接受具体的访问者来访
- **ConcreteElement：** 实现Element的accept操作，作为被访问者
- **ObjectStructure：** 能枚举元素；可提供一个高层接口以允许访问者访问其元素；可以是一个复合（组合模式）或是一个集合

访问者模式

- 例：为职员类添加新行为“增加假期”

```
class Element {
public:
    virtual void accept(Visitor
*visitor){};
};

class Employee : public Element {
public:
    string name;
    double income;
    int vacationDays;

    Employee(string name, double
income, int vacationDays) {
        this->name = name;
        this->income = income;
        this->vacationDays =
vacationDays;
    }

    void accept(Visitor *visitor) {
        visitor->visit(this);
    }
};
```

```
class Visitor {
public:
    virtual void visit(Element
*element){};
};

class IncomeVisitor : public Visitor {
public:
    void visit(Element *element) {
        Employee *employee = ((Employee
*)element);
        employee->income *= 1.10;
        cout << employee->name << "'s new
income: " << employee->income << endl;
    }
};

class VacationVisitor : public Visitor {
public:
    void visit(Element *element) {
        Employee *employee = ((Employee
*)element);
        employee->vacationDays += 3;
        cout << employee->name << "'s new
vacation days: " << employee-
>vacationDays << endl;
    }
};
```

访问者模式

- 例：为职员类添加新行为“增加假期”

```
class Employees {
private:
    list<Employee *> employees;
public:
    void attach(Employee *employee) {
        employees.push_back(employee);
    }
    void detach(Employee *employee) {
        employees.remove(employee);
    }
    void accept(Visitor *visitor) {
        for(auto it : employees) {
            it->accept(visitor);
        }
    }
};
```

```
int main(int argc, char **argv) {
    Employees *e = new Employees();
    e->attach(new Employee("Tom", 25000.0, 14));
    e->attach(new Employee("Thomas", 35000.0, 16));
    e->attach(new Employee("Roy", 45000.0, 21));

    IncomeVisitor *v1 = new IncomeVisitor();
    VacationVisitor *v2 = new VacationVisitor();

    e->accept(v1);
    e->accept(v2);

    return 0;
}
```

访问者模式

适用场景

- 一个对象结构包含很多类对象，它们有不同的接口，想对这些对象实施一些依赖于其具体类的操作
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而想避免让这些操作“污染”这些对象的类
- 当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作

优点

- 不修改具体元素类，就可以增加新操作。主要是通过元素类的accept方法接受一个visitor对象来实现的

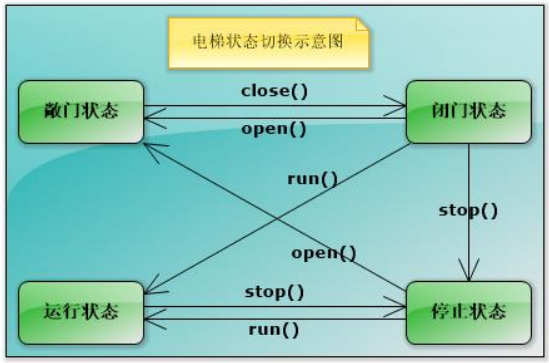
缺点

- 不易频繁增加元素类，每增加一个元素类，就要在Visitor接口中写一个针对该元素的方法，还要修改Visitor子类

状态模式

思考：电梯程序

```
class Context {
private:
    enum class State {
        Opening,
        Closing,
        Running,
        Stopping
    };
    State currentState;
public:
    Context() : currentState(State::Closing) {}

    void open() {
        if (currentState == State::Opening) {
            cout << "门已经是开启状态" << endl;
        } else if (currentState == State::Closing) {
            cout << "门开启电梯..." << endl;
            currentState = State::Opening;
        } else if (currentState == State::Running) {
            cout << "运行状态下无法开启门" << endl;
        } else if (currentState == State::Stopping) {
            cout << "电梯已停止，可以开启门" << endl;
            currentState = State::Opening;
        }
    }

    void close() {
        if (currentState == State::Opening) {
            cout << "电梯门关闭..." << endl;
            currentState = State::Closing;
        } else if (currentState == State::Closing) {
            cout << "门已经是关闭状态" << endl;
        } else if (currentState == State::Running) {
            cout << "电梯运行中，无法关闭门" << endl;
        } else if (currentState == State::Stopping) {
            cout << "电梯已停止，门已关闭" << endl;
        }
    }
```

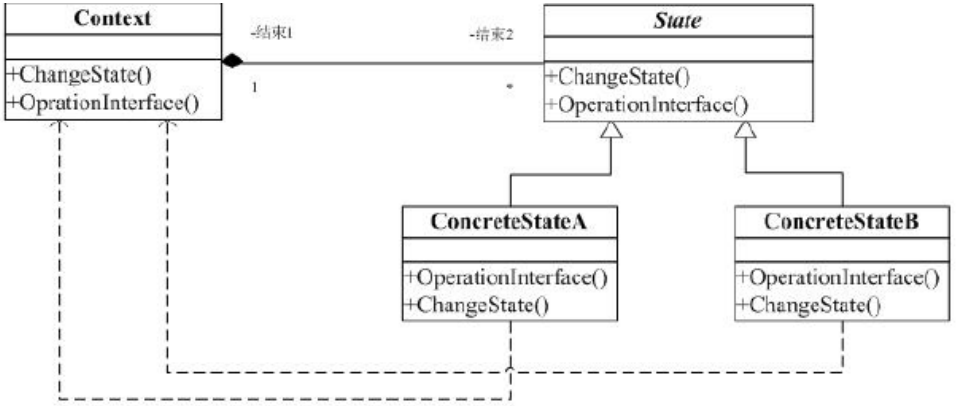



```
int main() {
    Context context;
    context.open();
    context.close();
    context.run();
    context.stop();
    return 0;
}
```

```
    void run() {
        if (currentState == State::Opening) {
            cout << "无法在开启状态下运行电梯" << endl;
        } else if (currentState == State::Closing) {
            cout << "电梯开始上下运行..." << endl;
            currentState = State::Running;
        } else if (currentState == State::Running) {
            cout << "电梯已经在运行中" << endl;
        } else if (currentState == State::Stopping) {
            cout << "电梯已停止，无法运行" << endl;
        }
    }

    void stop() {
        if (currentState == State::Opening) {
            cout << "无法在开启状态下停止电梯" << endl;
        } else if (currentState == State::Closing) {
            cout << "电梯已停止" << endl;
            currentState = State::Stopping;
        } else if (currentState == State::Running) {
            cout << "电梯停止运行" << endl;
            currentState = State::Stopping;
        } else if (currentState == State::Stopping) {
            cout << "电梯已经是停止状态" << endl;
        }
    }
};
```

状态模式

- 问题：一个对象有多种状态，在不同状态下，对象可以有不同行为
  - 采用if语句编写难以维护
- 状态（State）模式允许一个对象在其内部状态发生改变时改变其行为，使这个对象看上去就像改变了它的类型一样



- **Context：** 客户感兴趣的接口；维护一个ConcreteState子类实例，该实例定义了当前状态
- **State：** 封装与Context的一个特定状态相关的行为接口
- **ConcreteState subclasses：** 实现一个与Context的一个状态相关的行为

❑ 例：电梯程序

```
class LiftState {
protected:
    Context* context;
public:
    void setContext(Context* context)
{
        this->context = context;
    }
    virtual void open() = 0;
    virtual void close() = 0;
    virtual void run() = 0;
    virtual void stop() = 0;
    virtual ~LiftState() {}
};
```

```
int main() {
  Context context;
  context.setLiftState(new ClosingState());

  context.open();
  context.close();
  context.run();
  context.stop();

  return 0;
}
```

```
class Context {
public:
    LiftState* liftState;
    void setLiftState(LiftState* liftState) {
        if (this->liftState != nullptr) {
            delete this->liftState;
        }
        this->liftState = liftState;
        this->liftState->setContext(this);
    }
    void open() {
        liftState->open();
    }
    void close() {
        liftState->close();
    }
    void run() {
        liftState->run();
    }
    void stop() {
        liftState->stop();
    }
    ~Context() {
        delete liftState;
    }
};
```

❑ 例：电梯程序

```
//ConcreteState
class OpenningState : public LiftState {
public:
    void open() {
        cout << "门开启电梯..." << endl;
    }
    void close() {
        context->setLiftState(new ClosingState());
        context->liftState->close();
    }
    void run() {
        //开门状态下不能运行！
    }
    void stop() {
        //开门状态下，按停止没效果
    }
};
```

```
//运行状态
class RunningState : public LiftState {
public:
    void open() {
        //运行状态，按“开门”按钮没效课
    }
    void close() {
        //运行状态按“关门”按钮没效果
    }
    void run() {
        cout << "电梯上下运行..." << endl;
    }
    void stop() {
        context->setLiftState(new StoppingState());
        context->liftState->stop();
    }
};
```

```
//关门状态
class ClosingState : public LiftState {
public:
    void open() {
        context->setLiftState(new OpenningState());
        context->liftState->open();
    }
    void close() {
        cout << "电梯门关闭..." << endl;
    }
    void run() {
        context->setLiftState(new RunningState());
        context->liftState->run();
    }
    void stop() {
        context->setLiftState(new StoppingState());
        context->liftState->stop();
    }
};
```

```
//停止状态
class StoppingState : public LiftState {
public:
    void open() {
        context->setLiftState(new OpenningState());
        context->liftState->open();
    }
    void close() {
        //do nothing
    }
    void run() {
        context->setLiftState(new RunningState());
        context->liftState->run();
    }
    void stop() {
        cout << "电梯停止了..." << endl;
    }
};
```

上机安排/作业八

- 5月20日 13:00—15:00 理一1235机房

- 上机安排
  - 期中考试题目讲解
  - 第七次作业讲解
  - 大作业说明与答疑

- 作业八
  - 时间：5月20日 12:00—5月30日 23:59
  - 从教学网“课程作业”栏目下载“作业八”文档，完成后在教学网提交
    - 可提交手写拍照版本

谢谢

欢迎在课程群里填写问卷反馈