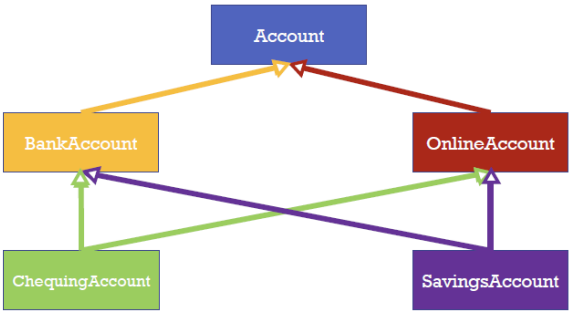
Jerry Xing A01354731 Set 3G

CASTING类型转换

18(22.7)

对象的类型在编译时定义，执行期间无法更改。类型转换操作符 dynamic_cast, const_cast, static_cast, reinterpret_cast



Upcasting: derived class 派生类转为 base class 基类：只要不存在歧义总是可行，与 polymorphism 多态性同义

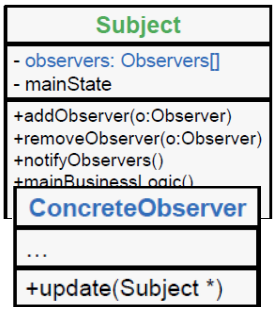
```
ChequingAccount c;
Account * accountPtr = dynamic_cast<Account *>(&c);
```

Downcasting：基类转派生类：实际所指对象非派生类时是未定义行为，`dynamic_cast<A*>(b)`将在运行时检查b是否真的指向A

```
SavingsAccount s;
OnlineAccount * oaPtr = dynamic_cast<OnlineAccount *>(&s); // 指针向上转型
SavingsAccount * saPtr = dynamic_cast<SavingsAccount *>(oaPtr); // 指针向下转型
```

Cross-casting：基类转 sibling class 兄弟类

```
ChequingAccount c;
BankAccount * baPtr = dynamic_cast<BankAccount *>(&c); // 指针向上转型
OnlineAccount *oaPtr = dynamic_cast<OnlineAccount *>(baPtr); // 指针交叉转型
```



动态转换：执行一个运行时测试，已确定被转换的对象是否具有目标类型或其子类型，仅当类多态时才能用(至少一个虚函数)，否则编译错误(最小成本变成多态可以创建一个空实现虚destructor析构：`virtual ~A() {}`)

静态转换：编译时转换，不检查，可用于指针和非多态类型(整数、浮点数)

Enums枚举

```
enum colour : int{RED, BLUE = 5}; /*可以指定底层值*/ int n = BLUE; /*n = 5*/
enum class Color : int{RED, BLUE}; /*默认是Int所以: int可省略*/
int b = static_cast<int>(Color::BLUE); /*作用域枚举禁止隐式转换*/
```

里氏替换(LSP)：子类应用不应改变父类语义

Design patterns 设计模式：优点：Design patterns provide reusable, abstract, language-agnostic, and proven solutions, improving code design quality and communication efficiency among developers.

缺点：Design patterns may increase system complexity, introduce additional constraints, lead to pattern overload, and do not directly result in code reuse.

分类：Behavioural行为:专注对象通信交互 communication and interaction between objs Structural结构: 专注类和对象组合 combine to form structures Creational创建:实例化类 class instantiation

Singleton 单例(图)：创建型，希望确保类A只有一个实例a被创建 | 1. 首次使用时才实例化(lazy init) 2. 私有构造函数，禁止外部new 3. 提供公共的get_instance方法返回唯一静态示例 4.其它程序可以访问该对象

```
// 单例模式
class singleton{
public:
    static singleton& get_instance(){// 允许通过公共接口访问单例
        static singleton instance;   // static 保证只存在一个单例实例
        return instance;             // 在第一次使用时才进行实例化
    }
private:
    int test_value;
    singleton() {} // 将构造函数设为私有，禁止外部直接构造对象
public:
    singleton(singleton const&) = delete;         // 禁止拷贝构造单例对象
    void operator=(singleton const&) = delete;    // 禁止对单例对象进行拷贝赋值
    int get_value() { return test_value++; }
};
```

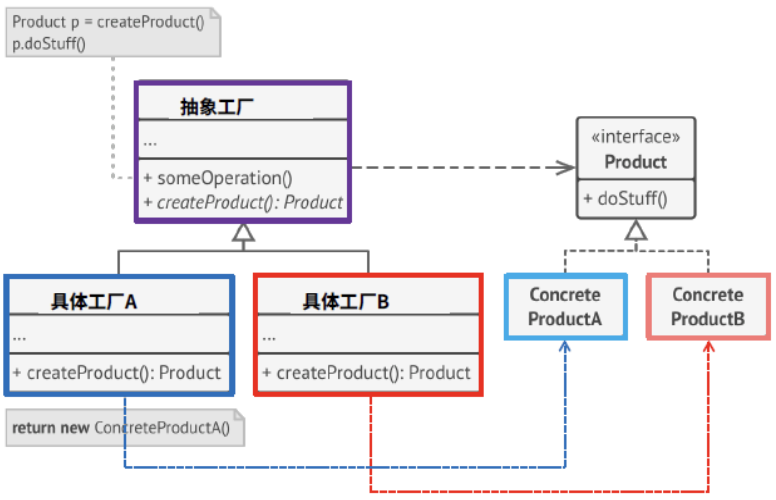
Observer 观察者(图): 行为型，Subject可以拥有任意数量Observer，Observer在Sub改变时通知

Wrapper 包装器：包含一个实例的方法，但增加额外行为，最终仍调用实例方法(而不是继承并修改方法)。下面三个都是结构型

Proxy 代理：结构型，控制对被包装对象的访问。优点：The Proxy pattern controls access and extends behaviour before or after service invocation without modifying the original service, encapsulating expensive objects while adhering to the Open–Closed Principle. 缺点：Introducing multiple proxies or placing excessive logic in a proxy can increase system complexity and potentially slow down service response time.

Facade 外观：结构型，为复杂API/子系统提供简单接口。如，客户端想执行操纵a，a需要类A、B的先后操作，但是不希望与AB耦合，则将调用隐藏在外观类C中，通过C提供调用AB两个方法的c方法来实现AB操作。优点：The Facade pattern provides a simple interface to encapsulate a complex system, reducing coupling, hiding details, and supporting layering or partial use of functionality. 缺点: A Facade can grow large and complex, and may become a central point of coupling in the system.

Bridge 桥接：结构型，将大型类或coupled耦合类拆分为abs抽象和impl实现。优点: The Bridge pattern is used to split a large, highly variable class into independent abstraction and implementation hierarchies so each dimension can evolve independently, even at runtime, while following the Open–Closed Principle. 缺点: If the dimensions are not truly independent, the Bridge pattern can make the design overly complex and blur the distinction between abstraction and implementation.

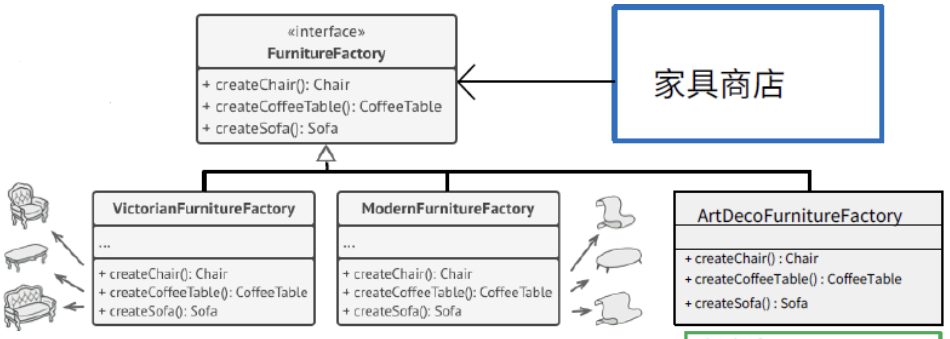


Factory 工厂(图)：创建型，将"对象创建"与"对象使用"decouple解耦，让客户端只依赖抽象，而不依赖具体类，避免high level module高层模块dependent on依赖低层模块(DIP依赖倒置)。

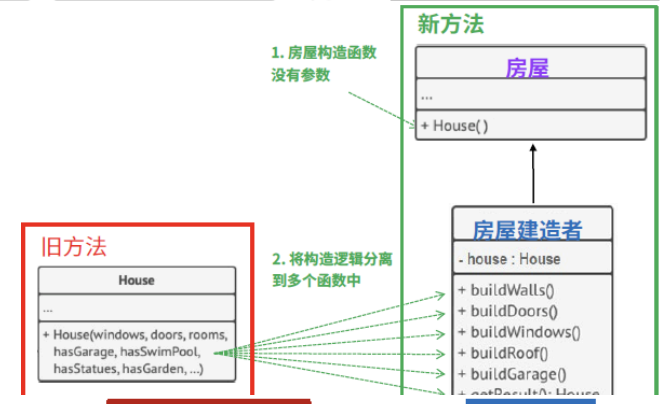
优点：The Factory Pattern separates object creation from usage, allowing subclasses to decide which objects to create while adhering to SOLID principles and accommodating change. 缺点：The pattern increases the number of classes and abstractions,

which can make the codebase more complex and sometimes overly engineered for small changes.

Abs Factory 抽象工厂(图): 创建型，创建一组相关的对象，而无需指定具体类。工厂模式中，每个工厂创建单一产品。抽象工厂中，每个工厂创建一个group of related products相关产品的组合（如，产品有不同变体）。优点：The Abstract Factory Pattern is used to create families of related objects without specifying concrete classes, while adhering to SRP, OCP, LSP, and DIP and supporting future product families. 缺点：It introduces many classes and interfaces, increasing complexity, and may feel over-engineered when only minor changes to object creation are needed.



Builder 构建器(图): 创建型。如果一个对象的构造过程非常复杂，可以将其分离到一个独立的类中，确保Single Responsibility Principle.单一职责原则。优点：The Builder Pattern separates complex object construction from the product itself, reducing constructor overloads and enabling flexible composition of parts while following the Single Responsibility Principle. 缺点：It introduces additional classes and complexity, and is unsuitable for products that cannot be decomposed into independent parts.

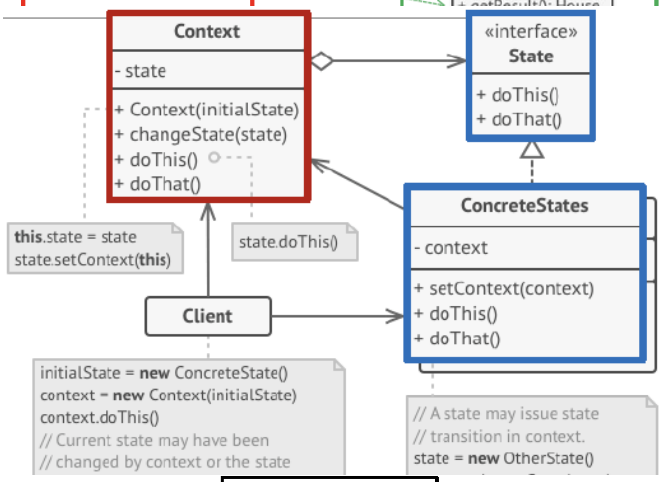


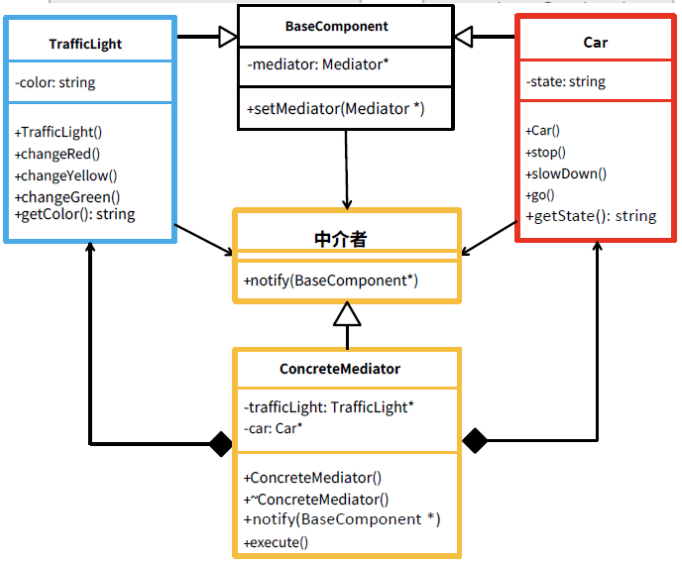
Chain of Res 责任链: 行为型。如果一个/一组对象需要经历不同的处理步骤，建立一系列实现相同接口、唯一的Handler，分别处理其中一个请求。优点：The Chain of Responsibility pattern lets multiple handlers process a request in sequence, enabling flexible ordering, decoupling senders from handlers, and supporting SRP and OCP. 缺点：It adds more classes to maintain, and requests may go unhandled if the handler order is misconfigured.

Lazy Init 惰性初始化: 行为型。除非确实使用，否则不初始化对象。优点：Dedicate resources (memory / processing) on a on-demand basis. 缺点：Not a good choice if the obj needs to be init every time it is accessed

Strategy 策略: 行为型。将可互换的算法封装成独立的对象，使算法可以在运行时自由替换，而不影响客户端。优点：The Strategy pattern encapsulates interchangeable algorithms, allowing them to be swapped at runtime, isolating algorithm implementations from the code that uses them and adhering to OCP. 缺点：It can be overkill when few algorithms rarely change, and it requires the client to be aware of and choose among available strategies.

State 状态(图): 行为型。对象的状态是指在程序执行期间对象的值和行为。状态模式提供了一种模块化的方式来添加状态并控制状态转换，以将状态与对象解耦。当一个对象的行为严重依赖状态时采用该模式。具有多个状态的对象称为Context。优点：The State pattern is used when an object's behaviour depends heavily on its internal state, allowing states to be added or changed without large conditional logic while adhering to SRP and OCP. 缺点：The State pattern can introduce many classes and complex state transitions, making the design more challenging to maintain when there are few states or infrequent state changes.与策略的不同：**Strategy** 是由客户端在运行时主动选择、彼此独立且互不知晓的算法切换，不基于有限状态机，策略通常不持有 Context；而 **State** 是由 Context 根据有限状态机自动切换、状态之间可相互认知且可持有 Context，从而在状态变化时整体改变对象行为。**Strategy** is selected explicitly by the client at runtime to switch between independent algorithms that do not know about each other, are not based on a finite state machine, and typically do not hold a reference to the Context. In contrast, the State is switched automatically by the Context, based on a finite state machine, allowing states to be aware of one another and to hold a reference to the Context, thereby changing the object's overall behaviour as its state changes.





Mediator 中介者(图): 行为型。当不同对象之间的通信模式变得混乱且耦合时使用。优点：The Mediator pattern centralizes communication between objects to reduce chaotic, tightly coupled interactions while keeping components unaware of each other. 缺点：The Mediator can become overly complex and turn into a "God object," acting as a central point of coupling.

Decorator 装饰器: 结构型，通过Wrapping包装的方式，在不修改原有类的情况下动态地为对象添加额外行为。不是继承原对象，因为组合越多→子类越多，不符合OCP开闭原则。优点：The Decorator pattern allows behaviour to be added or removed at runtime by composing objects, supporting the Single Responsibility and Open–Closed principles without relying on inheritance. 缺点：The Decorator pattern can be hard to understand, debug, and manage when many decorators are stacked, mainly since behaviour may depend on the order of decoration. 参考代码

```
Component* d = new WheatDonut();
d = new Sprinkles(d);
d = new Gummies(d);
```

```
// （观察者）观察者接口
interface Observer {update();}
// 被观察者接口
interface Subject {attach(o : Observer);detach(o : Observer);notify();}
// 具体观察者
class User implements Observer {
    update() {/*receive notification*/}
}
// 具体被观察者
class NewsPublisher implements Subject {
    observers;
    attach(o) {/*add observer*/}
    detach(o) {/*remove observer*/}
    notify() {/*for each o -> o.update()*/}
}
```

```
// （外观）子系统
class CPU {start() {/*start cpu*/}}
class Memory {load() {/*load memory*/}}
class Disk {read() {/*read disk*/}}
// 外观
class ComputerFacade {
    CPU cpu; Memory memory; Disk disk;
    start() {cpu.start();memory.load(); disk.read();}
}
```

```
// （工厂）产品接口
interface Food {produce();}
// 具体产品
class Bread implements Food {produce() {/*produce bread*/}}
class Milk implements Food {produce() {/*produce milk*/}}
// 工厂抽象
abstract class FoodFactory {createFood() : Food}
// 具体工厂
class BreadFactory extends FoodFactory {createFood() return new Bread();}
class MilkFactory extends FoodFactory {createFood() return new Milk();}
// 客户端
class FoodProductionSystem {
    FoodFactory factory;
    FoodProductionSystem(factory) {this.factory = factory;}
    produceFood() { Food f = factory.createFood();f.produce();}
}
```

```
// （构建器）产品
class House {walls; roof; door;}
// 构建者接口
interface HouseBuilder {
    buildWalls(); buildRoof();
    getHouse() : House;
}
// 具体构建者
class WoodenHouseBuilder implements HouseBuilder {
    house;
    buildWalls() {/*wood walls*/}
    buildRoof() {/*wood roof*/}
    getHouse() {return house;}
}
// 指挥者
class Director {
    build(builder : HouseBuilder) {
        builder.buildWalls(); builder.buildRoof();
        return builder.getHouse(); }
}
```

```
// 惰性初始化
class Database {
    static instance;
    static getInstance() {
        if (instance == null) {instance = new Database();}
        return instance;
    }
}
```

```
// （代理）服务接口
interface Service {request();}
// 真实服务
class RealService implements Service {request() {/*do real work*/}}
// 代理
class ServiceProxy implements Service {
    RealService service;
    ServiceProxy(service) {this.service = service;}
    request() {/*check access / cache / log*/service.request();}
}
```

```
// （桥接）实现接口
interface Renderer {render();}
// 具体实现
class VectorRenderer implements Renderer {render() {/*vector render*/}}
class RasterRenderer implements Renderer {render() {/*raster render*/}}
// 抽象
abstract class Shape {
    Renderer renderer;
    Shape(renderer) {this.renderer = renderer;}
    draw();
}
// 细化抽象
class Circle extends Shape {draw() {renderer.render();}}
// （抽象工厂）产品接口
interface Food {produce();}
interface Package {pack();}
// 具体产品（食品工厂族 A）
class Bread implements Food {produce() {/*produce bread*/}}
class BreadBox implements Package {pack() {/*box bread*/}}
// 具体产品（食品工厂族 B）
class Milk implements Food {produce() {/*produce milk*/}}
class MilkBottle implements Package {pack() {/*bottle milk*/}}
// 抽象工厂
interface FoodFactory {createFood() : Food; createPackage() : Package;}
// 具体工厂
class BakeryFactory implements FoodFactory {
    createFood() return new Bread(); createPackage() return new BreadBox();}
class DairyFactory implements FoodFactory {
    createFood() return new Milk(); createPackage() return new MilkBottle();}
// 客户端
class FoodProductionSystem {
    FoodFactory factory;
    FoodProductionSystem(factory) {this.factory = factory;}
    produce() {
        Food f = factory.createFood();
        Package p = factory.createPackage();
        f.produce(); p.pack();
    }
}
```

```
// （责任链）处理者接口
interface Handler { setNext(h : Handler); handle(request);}
// 抽象处理者
abstract class BaseHandler implements Handler {
    next; setNext(h) {next = h;}
    handle(request) {if (next) next.handle(request);}
}
// 具体处理者
class AuthHandler extends BaseHandler {
    handle(request) {
        if (request.auth) {/*pass*/}
        else {/*reject*/}
        super.handle(request);
    }
}
class LogHandler extends BaseHandler {
    handle(request) {
        /*log request*/
        super.handle(request);
    }
}
```

```
// （状态）接口
interface State {handle();}
// 具体状态
class OnState implements State {handle() {/*device on*/}}
class OffState implements State {handle() {/*device off*/}}
// 上下文
class Device {
    State state;
    Device(state) {this.state = state;}
    setState(state) {this.state = state;}
    press() {state.handle();}
}
// （中介者）接口
interface Mediator {notify(sender, event);}
// 同事接口
interface Colleague {action();}
// 具体同事
class Button implements Colleague {
    Mediator mediator; Button(m) {mediator = m;}
    action() {mediator.notify(this, "click");}
}
class TextBox implements Colleague {
    action() {/*update text*/}
}
// 具体中介者
class DialogMediator implements Mediator {
    Button button; TextBox textbox;
    notify(sender, event) {
        if (sender == button && event == "click") textbox.action();
    }
}
```

```
// （策略）策略接口
interface Strategy {execute();}
// 具体策略
class QuickStrategy implements Strategy {execute() {/*fast way*/}}
class SafeStrategy implements Strategy {execute() {/*safe way*/}}
// 上下文
class Context {
    Strategy strategy;
    Context(strategy) {this.strategy = strategy;}
    run() {strategy.execute();}
}
```

```
// （装饰器）组件接口
interface Coffee {cost();}
// 具体组件
class SimpleCoffee implements Coffee {cost() {return 5;}}
// 装饰器抽象
abstract class CoffeeDecorator implements Coffee {
    Coffee coffee;
    CoffeeDecorator(coffee) {this.coffee = coffee;}
    cost() {return coffee.cost();}
}
// 具体装饰器
class MilkDecorator extends CoffeeDecorator {cost() {return coffee.cost() + 2;}}
class SugarDecorator extends CoffeeDecorator {cost() {return coffee.cost() + 1;}}
```