**1. 单一职责原则（SRP）:**一个类只负责一个职责。

|  |  |
| --- | --- |
| 错误设计：OrderManager类同时负责订单处理和日志记录。  **class OrderManager:**  def process\_order(self, order):  # 处理订单  self.log\_order(order)  def log\_order(self, order):  print(f"Order {order.id} logged.") | 正确设计：将日志记录职责抽离到单独的类。  **class OrderManager:**  def \_\_init\_\_(self, logger):  self.logger = logger  def process\_order(self, order):  # 处理订单  self.logger.log(order)  **class Logger:**  def log(self, order):  print(f"Order {order.id} logged.") |
|  |  |

**2. 开闭原则（OCP）:**对扩展开放，对修改封闭。

|  |  |
| --- | --- |
| 错误设计：在现有的支付类中直接添加新的支付方式。  class PaymentProcessor:  def process\_payment(self, payment\_type):  if payment\_type == "credit\_card":  print("Processing credit card payment.")  elif payment\_type == "paypal":  print("Processing PayPal payment.") | 正确设计：使用抽象类扩展新的支付方式。  class PaymentProcessor:  def process\_payment(self, payment):  payment.pay()  class CreditCardPayment:  def pay(self):  print("Processing credit card payment.")  class PayPalPayment:  def pay(self):  print("Processing PayPal payment.") |
|  |  |

**3.里氏替换原则（LSP）:**子类可以替换父类。

|  |  |
| --- | --- |
| 错误设计：Square继承Rectangle但改变了行为。  class Rectangle:  def \_\_init\_\_(self, width, height):  self.width = width  self.height = height  def area(self):  return self.width \* self.height  class Square(Rectangle):  def \_\_init\_\_(self, side):  super().\_\_init\_\_(side, side) | 正确设计：将矩形和正方形分离。  class Shape:  def area(self):  pass  class Rectangle(Shape):  def \_\_init\_\_(self, width, height):  self.width = width  self.height = height  def area(self):  return self.width \* self.height  class Square(Shape):  def \_\_init\_\_(self, side):  self.side = side  def area(self):  return self.side \* self.side |
|  |  |

**4. 依赖倒置原则（DIP）:**高层模块依赖于抽象，而不是具体实现。

|  |  |
| --- | --- |
| 错误设计：OrderProcessor直接依赖于具体的EmailNotifier。  class EmailNotifier:  def send(self, message):  print(f"Email sent: {message}")  class OrderProcessor:  def \_\_init\_\_(self):  self.notifier = EmailNotifier()  def process\_order(self):  self.notifier.send("Order processed.") | 正确设计：通过接口注入依赖。  class Notifier:  def send(self, message):  pass  class EmailNotifier(Notifier):  def send(self, message):  print(f"Email sent: {message}")  class OrderProcessor:  def \_\_init\_\_(self, notifier):  self.notifier = notifier  def process\_order(self):  self.notifier.send("Order processed.") |
|  |  |

例二

|  |  |
| --- | --- |
|  |  |

**5. 接口分离原则（ISP）:**不应该强迫一个类实现不需要的接口。

|  |  |
| --- | --- |
| 错误设计：一个接口包含了所有方法。  class Worker:  def work(self):  pass  def eat(self):  pass | 正确设计：将接口分离。  class Workable:  def work(self):  pass  class Eatable:  def eat(self):  pass  class Worker(Workable, Eatable):  def work(self):  print("Working")  def eat(self):  print("Eating") |
|  |  |

**6. 发布复用等价性原则（REP）:**模块的发布和重用是等价的。

|  |
| --- |
| 将日志功能单独封装成模块，可以在不同的项目中重用。  # log\_module.py  class Logger:  def log(self, message):  print(message) |

**7. 共同封装原则（CCP）:**需要一起修改的类应该放在同一个模块中。

|  |
| --- |
| 用户登录系统的类应封装在一个模块中。  # authentication\_module  class UserValidator:  def validate(self, username, password):  pass  class SessionManager:  def create\_session(self, user):  pass |

**8. 共同复用原则（CRP）:**一个模块只包含一起被复用的类。

|  |
| --- |
| 将支付相关类封装在支付模块中，而不包含其他无关功能。  # payment\_module  class CreditCardPayment:  def pay(self):  pass  class PayPalPayment:  def pay(self):  pass |

**UML图**

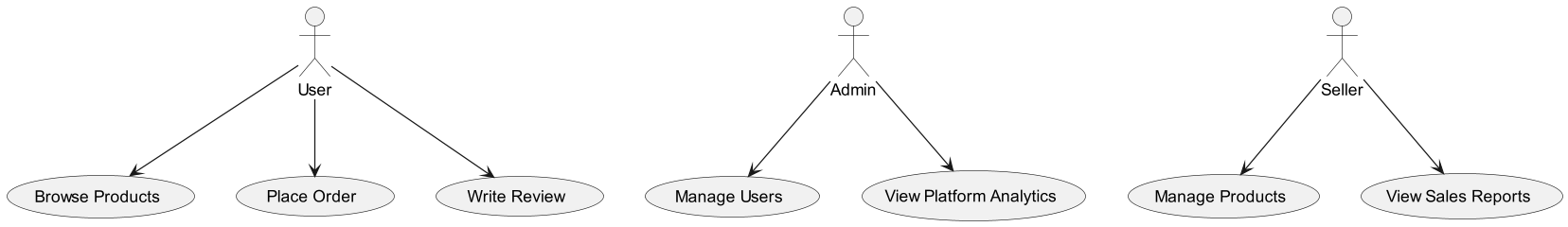
**1.用例图 (Use Case Diagram)**

包含的元素：

* 参与者 (Actor)：系统的用户或其他外部系统，分为主角和次要角色。
* 用例 (Use Case)：系统提供的功能或服务，用椭圆表示。
* 关联关系 (Association)：参与者与用例之间的关系，用直线连接。
* 扩展 (Extend)：一种可选的用例行为，表示用例可以被扩展，触发条件明确。
* 包含 (Include)：通用的用例行为，表示用例包含了另一个用例。
* 系统边界 (System Boundary)：定义系统范围，用一个矩形框表示。

与其他图的区别：

* 目标：用例图关注系统的功能需求以及用户与系统的交互，而其他图多描述实现细节。
* 区别：用例图描述的是功能需求，是高层次的抽象，不涉及内部实现。



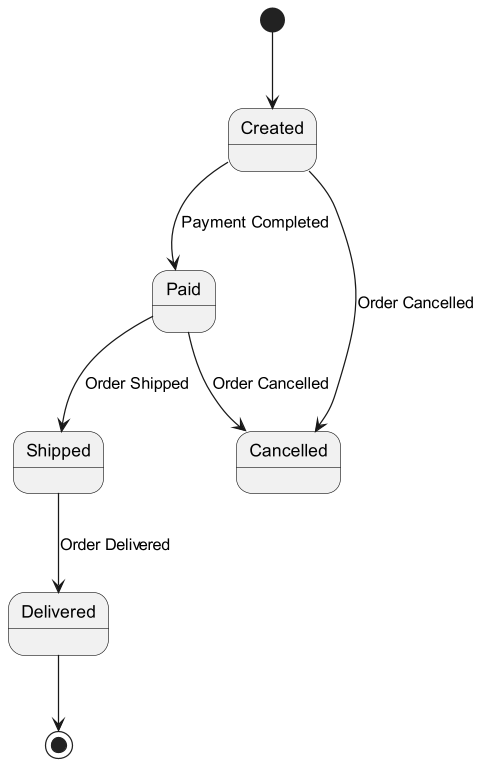
**2. 状态图 (State Diagram)**

包含的元素：

* 状态 (State)：系统的某个状态，用圆角矩形表示。
* 初始状态 (Initial State)：系统的起始状态，用黑色实心圆表示。
* 结束状态 (Final State)：系统的终止状态，用带圆圈的实心圆表示。
* 状态转换 (Transition)：从一个状态到另一个状态的路径，用箭头表示，通常附带触发事件和条件。
* 事件 (Event)：触发状态转换的操作或条件。

与其他图的区别：

* 目标：状态图描述对象或系统在生命周期中的状态变化过程。
* 区别：状态图专注于单个对象的状态变化，而活动图描述流程，顺序图描述对象之间的交互。



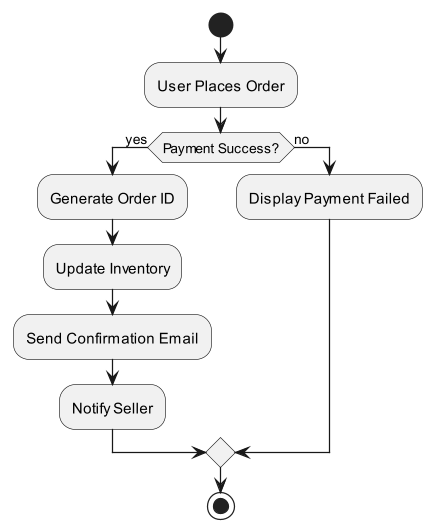
**3. 活动图 (Activity Diagram)**

包含的元素：

* 活动 (Activity)：代表某个步骤或操作，用矩形表示。
* 决策节点 (Decision Node)：用于分支流程，用菱形表示。
* 合并节点 (Merge Node)：多个分支合并成一条流程，用菱形表示。
* 并行节点 (Fork Node)：分支出多条并行路径，用实心水平线表示。
* 汇聚节点 (Join Node)：多条并行路径合并成一条，用实心水平线表示。
* 起始节点 (Initial Node)：流程的起始点，用黑色实心圆表示。
* 结束节点 (Final Node)：流程的终止点，用带圆圈的实心圆表示。
* 对象流 (Object Flow)：表示活动之间的数据传递。

与其他图的区别：

* 目标：活动图描述系统的动态行为，主要用于建模业务流程或工作流。
* 区别：活动图着眼于流程逻辑，而状态图关注状态变化。



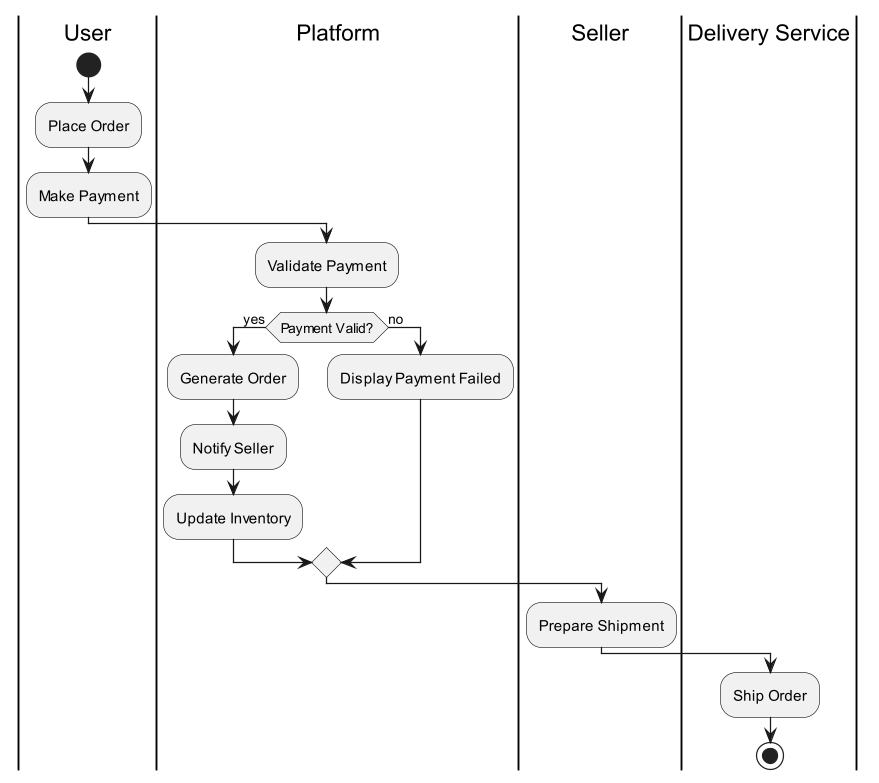
**4. 泳道图 (Swimlane Diagram)**

包含的元素：

* 泳道 (Swimlane)：将活动划分到不同的角色或部门中，每个泳道代表一个参与方。
* 活动 (Activity)：泳道中的具体任务或操作。
* 箭头 (Arrow)：表示活动之间的流程。

与其他图的区别：

* 目标：泳道图是活动图的变体，重点关注责任分配，展示不同角色之间的活动交互。
* 区别：泳道图突出了角色或部门，活动图则不强调这一点。



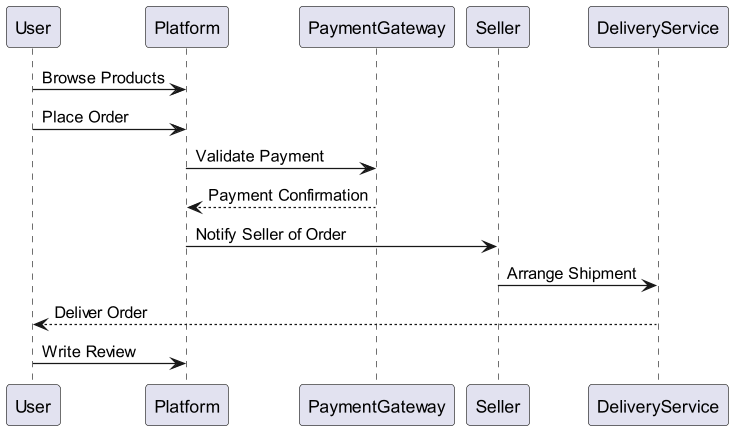
**5. 顺序图 (Sequence Diagram)**

包含的元素：

* 对象 (Object)：用矩形表示，放在图的顶部。
* 生命线 (Lifeline)：表示对象的生命周期，用垂直虚线表示。
* 消息 (Message)：对象之间的交互，用箭头表示，有同步、异步、返回消息等。
* 激活条 (Activation Bar)：表示对象在某个时间段内的活动，用细长的矩形表示。
* 创建与销毁：表示对象的创建（箭头指向生命线）或销毁（"X"符号）。

与其他图的区别：

* 目标：顺序图描述系统中对象之间的交互顺序。
* 区别：顺序图注重时间顺序，而类图描述静态结构，活动图描述流程。



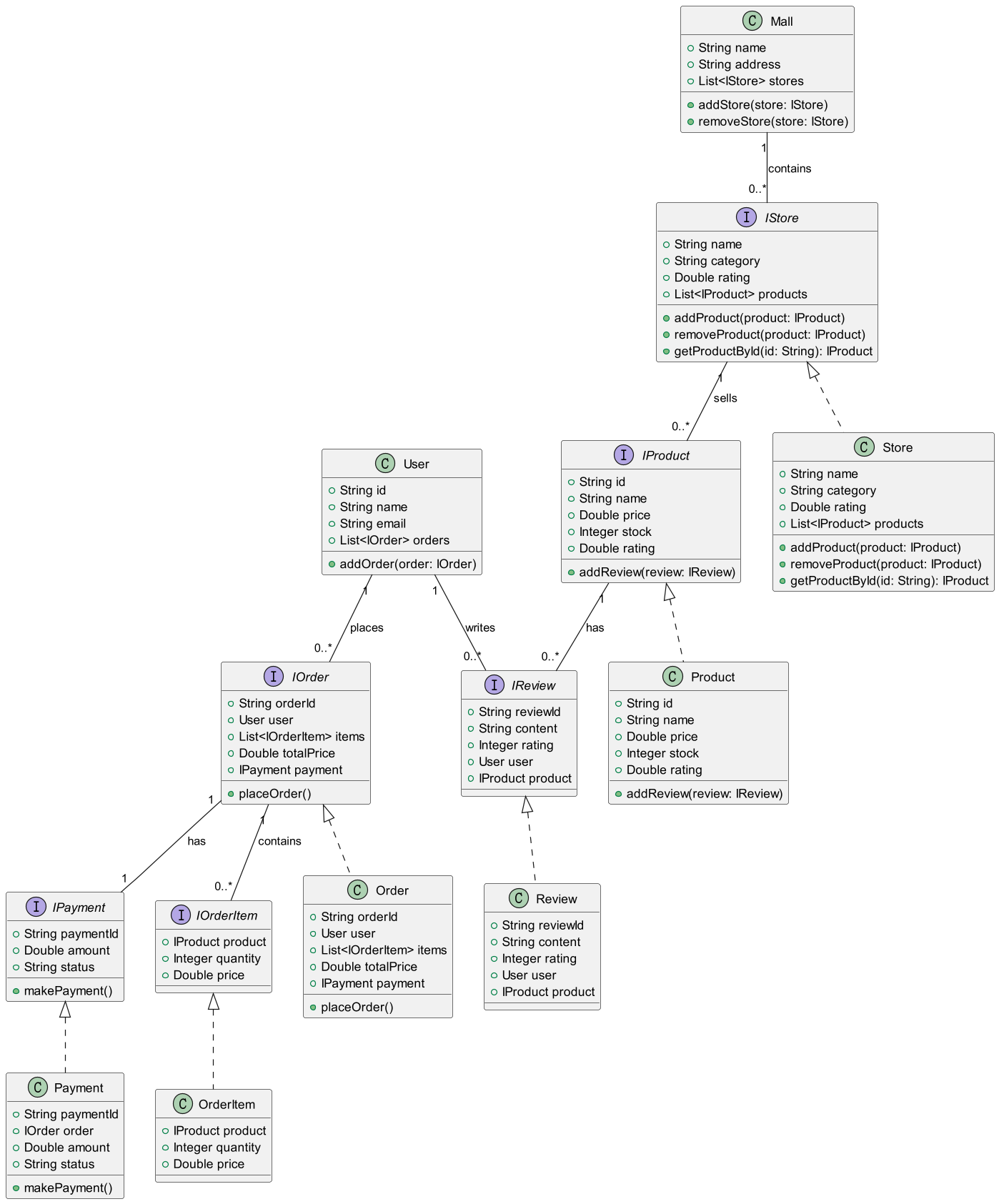
**6. 类图 (Class Diagram)**

包含的元素：

* 类 (Class)：用矩形表示，分为类名、属性、方法三部分。
* 关系 (Relationships)：
* 关联 (Association)：两个类之间的普通关系，用实线表示。
* 继承 (Inheritance)：表示父类和子类之间的关系，用带空心箭头的实线表示。
* 依赖 (Dependency)：表示类之间的依赖，用带虚线箭头表示。
* 聚合 (Aggregation)：表示整体与部分的关系，用空心菱形和实线表示。
* 组合 (Composition)：表示强依赖的整体与部分关系，用实心菱形和实线表示。

与其他图的区别：

* 目标：类图描述系统的静态结构，展示类及其关系。
* 区别：类图是静态建模图，而顺序图、活动图等是动态行为建模图。



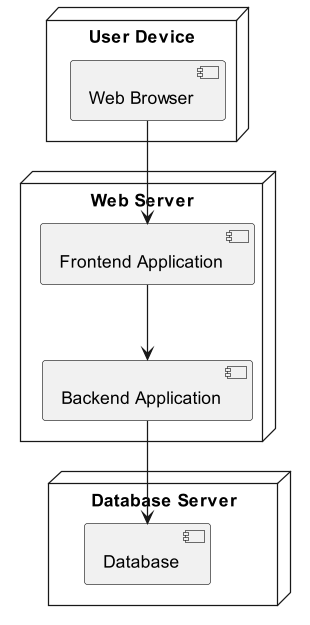
**7. 部署图 (Deployment Diagram)**

包含的元素：

* 节点 (Node)：硬件设备或运行环境，用立方体表示。
* 组件 (Component)：系统中的可部署单元，用矩形表示。
* 连接 (Association)：节点之间的通信路径，用带箭头的实线表示。

与其他图的区别：

* 目标：部署图描述系统的物理部署结构。
* 区别：部署图着重于硬件环境和运行组件，而类图描述逻辑结构，活动图描述流程。



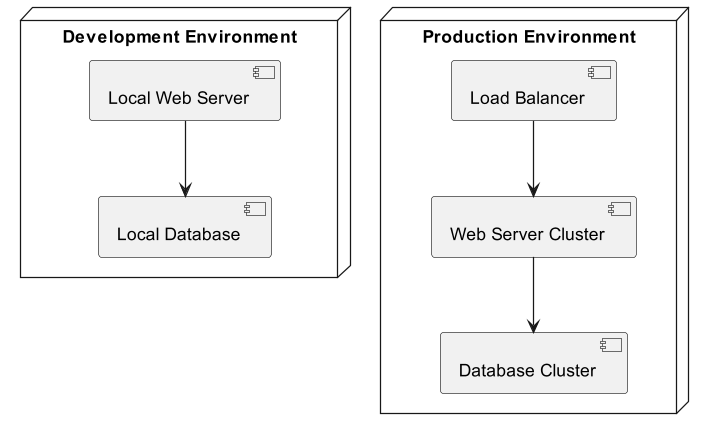
**8. 环境图 (Environment Diagram)**

包含的元素：

* 节点 (Node)：通常表示系统或环境的硬件或软件，用矩形或立方体表示。
* 连接 (Connections)：节点之间的通信方式，用线或箭头表示。
* 环境变量：特定的运行时配置信息。

与其他图的区别：

* 目标：环境图专注于运行环境中各组件的配置和连接。
* 区别：与部署图相似，但环境图更多关注软件配置（如操作系统、数据库连接等）。



**9.各图之间的区别和联系**

功能与需求：

用例图：从用户视角描述功能需求。

类图：描述功能的实现方式。

动态行为：

顺序图：描述对象之间的时间顺序交互。

活动图：描述流程逻辑。

状态图：描述对象状态变化。

静态结构：

类图：展示系统的逻辑结构。

部署图：展示系统的物理部署。

流程与责任：

活动图：展示业务流程。

泳道图：强调流程中角色的责任分配。