UML期末复习

1. 什么是UML? P8(中文教程)

统一建模语言（UML）是描述、构造和文档化系统制品的可视化语言

应用UML的三种方式：UML作为草图；UML作为蓝图（逆向工程，代码生成）；UML作为编程语言

UML和银弹思想：如果不具备良好的OO设计和编程技能，那么即使使用UML，也只能画出拙劣的设计

应用UML的三种透视图：概念透视图（用图来描述现实世界和关注领域中的事物）；规格说明透视图（用图来描述软件的抽象物或具有规格说明和接口的构件）；实现透视图（用图来描述特定技术中的软件实现）

不同透视图中“类”的含义：概念类（现实世界中的概念或事物）；软件类（软件构件在规格说明或实现透视图中的类）；实现类（特定OO语言中的类）

1. 什么是面向对象的分析与设计? P5

面向对象分析：强调在问题领域内发现和描述对象

面向对象设计：强调定义软件对象以及它们如何协作以实现需求

1. 什么是UP? RUP? P13

UP：一种流行的构造面向对象系统的迭代软件开发过程

RUP：对统一过程的详细精化，并且已经被广泛采纳

1. 什么是UP的阶段? 什么是UP科目? P25

UP阶段：

初始（构想、业务案例、范围、模糊评估）

细化（已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围、更为实际的评估）

构造（对遗留下来的风险较低和比较简单的元素进行迭代实现、准备部署）

移交（beta测试、部署）

科目：一个主题域中的一组活动

UP科目：

业务建模（领域模型制品，使应用领域中的重要概念的可视化）

需求（用以捕获功能需求和非功能需求的用例模型及其补充性的规格说明制品）

设计（设计模型制品，用于对软件对象进行设计）

本书关于UP阶段的组织：

初始阶段对应的章介绍需求分析的基本内容

迭代1介绍OOA/D基础和如何为对象分配职责

迭代2的重点是对象设计，特别介绍一些经常使用的“设计模式”

迭代3介绍各种主题，例如架构分析和框架设计

1. 为什么说初始阶段不是需求阶段? P37

初始阶段的目标并不是定义所有需求，或产生可信的预算或项目计划。大多数需求分析是在细化阶段进行的，并且伴以具有产品品质的早期编程和测试。

这是一个关键点，当人们叠加了固有的“瀑布”思维时会重蹈对UP项目的误解。UP不是瀑布，初始阶段作为UP的第一个阶段也不需要完成所有需求或建立可靠预算和计划。以上内容是在细化的过程中逐步完成的。

1. 什么是参与者? P47

参与者是任何具有行为的事物，可分为：主要参与者、协助参与者和幕后参与者。

1. 什么是场景?如何写场景?P47

场景是参与者和系统之间的一系列特定的活动和交互，也称为用例实例。场景是使用系统的一个特定情节或用例的一条执行路径。

1. 什么是用例? P47

用例是一组相关的成功和失败场景集合，用来描述参与者如何使用系统来实现其目标

[RUP]一组用例的实例，其中每个实例都是系统执行的一系列活动，这些活动产生了对某个参与者而言可观察的返回值

1. 什么是用例实现? P232

用例实现描述某个用例基于协作对象如何在设计模型中实现，用以提示我们在表示为用例的需求和满足需求的对象之间的联系

SSD、系统操作、交互图和用例实现

用例和用例实现

操作契约和用例实现

领域模型和用例实现

1. 如何找到概念类? P104

概念类是思想、事物或对象。更正式地讲，概念类可以从其符号、内涵和外延来考虑。找到概念类的三条策略：重用和修改现有的模型；使用分类列表；确定名词短语。

1. 何时使用描述类建模? P109 什么是数据类型类? P120 什么是限定关联? P192

A描述类包含描述其他事物的信息，在以下情况下需要增加描述类：

需要有关商品或服务的描述，独立于任何商品或服务的现有实例；

删除其所描述事物的实例后，导致信息丢失，而这些信息是需要维护的，但是被错误地与所删除的事物关联起来；

减少冗余或重复信息。

B 数据类型类：如果数据类型是具有属性和关联的新类型，将其表示为单独的概念类

C 限定关联：具有限定符，限定符用于从规模较大的相关对象集合中，依据限定符的键选择一个或多个对象。限定关联暗示了基于键对事物进行查找。受限关联减少了关联目标端的多重性，通常是由多变为一。

1. 什么是SSD? P128

系统顺序图表示，对于用例的一个特定场景，外部参与者产生的事件，其顺序和系统之内的事件。UML包含了顺序图作为表示法，以便能够阐述参与者的交互及参与者引发的操作。

1. SSD与用例之间的关系? P129

SSD展示了用例中一个场景的系统事件

1. GRASP 原则有哪些? 请准确理解它们的含义.（17、25章！重点！以书上内容为准，以下仅供参考）

|  |  |
| --- | --- |
| 模式名称 | 描述（问题/解决方案) |
| 信息专家模式 | 问题：对象设计和职责分配的一般原则是什么？  解决方案：将职责分配给拥有履行一个职责所必需信息的类－－即信息专家。（也就是将职责分配给一个类，这个类必须拥有履行这个职责所需要的信息。） |
| 创建者模式 | 问题：谁应该负责产生类的实例（对应于GoF设计模式系列里的“工厂模式”）  解决方案：如果符合下面的一个或多个条件，则将创建类A实例的职责分配给类B.  .类B聚合类A的对象。 .类B包含类A的对象。 .类B记录类A对象的实例。 .类B密切使用类A的对象。 .类B初始化数据并在创建类A的实例时传递给类A（类B是创建类A实例的一个专家)。  在以上情况下，类B是类A对象的创建者。 |
| 控制器模式 | 问题：谁处理一个系统事件？  解决方案：当类代表下列一种情况时，为它分配处理系统事件消息的职责。 .代表整个系统、设备或子系统（外观控制器）。 .代表系统事件发生的用例场景（用例或回话控制器）。 |
| 低耦合 | 问题：如何支持低依赖性以及增加重用性？  解决方案：分配职责时使（不必要的）耦合保持为最低。 |
| 高内聚 | 问题：如何让复杂性可管理？  解决方案：分配职责时使内聚保持为最高。 |
| 多态模式 | 问题：当行为随类型变化而变化时谁来负责处理这些变化？  解决方案：当类型变化导致另一个行为或导致行为变化时，应用多态操作将行为的职责分配到引起行为变化的类型。 |
| 纯虚构模式 | 问题：当不想破坏高内聚和低耦合的设计原则时，谁来负责处理这些变化？  解决方案：将一组高内聚的职责分配给一个虚构的或处理方便的“行为”类，它并不是问题域中的概念，而是虚构的事务，以达到支持高内聚、低耦合和重用的目的。 |
| 中介模式 | 问题：如何分配职责以避免直接耦合？  解决方案：分配职责给中间对象以协调组件或服务之间的操作，使得它们不直接耦合。 |
| 受保护变化模式 | 问题：如何分配职责给对象、子系统和系统，使得这些元素中的变化或不稳定的点不会对其他元素产生不利影响？  解决方案：找出预计有变化或不稳定的元素，为其创建稳定的“接口”而分配职责。 |

1. 什么是持久性对象? P447

持久性对象是指需要持久性存储的对象

1. 什么是框架? P447

框架：一组功能相关的可扩展对象。

框架的显著特征是，为核心和不变功能提供实现，并且包括了允许开发者插入或者扩展可变功能的机制。

框架是：

一组相关的类和接口相互协作，为逻辑子系统的核心和不变部分提供服务

包含具体的抽象类，这些类定义了需要遵循的接口、需要参与的对象交互以及其他不变式

通常要求框架的使用者去定义已有框架类的子类来利用、定制或扩展框架服务

包含既有抽象方法又有具体方法的抽象类

1. 什么是O/R映射? P447

持久性服务通常与关系数据库一起工作，此时也称为O-R映射服务，持久性服务将对象转换为记录，并将它们存入数据库；从数据库读取时，将记录转换成对象

1. 熟练掌握UML标记(关联,聚合,组合,实现,泛化,依赖等)

关联：类之间的关系，表示有意义和值得关注的连接（直线和导航性箭头）

聚合：模糊关联，不精确的暗示了整体-部分关系

组合：组成聚合，一种很强的整体-部分关系

实现：一个class类实现interface接口（可以是多个）的功能；实现是类与接口之间最常见的关系

泛化：在多个概念中识别共性和定义超类与子类关系的活动

依赖：可以视为耦合的另一版本

拥有提供者类型的属性

向提供者发送消息

接受提供者类型的参数

提供者是超类或接口

1. 熟练掌握各种图形的画法.(用例图,类图,顺序图,活动图,状态机图)
2. 什么是设计模式? 我们共学习了哪些设计模式? 请熟练掌握它们的用法.

Adapter模式

把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作，也就是说把接口不同而功能相同或相近的多个接口加以转换。

1. 类的Adapter模式的结构



1. 对象的Adapter模式的结构



注意两种结构的区别：主要就是Adaptee和Adapter的关系，一个为继承关系，一个为依赖关系。

使用Adapter模式的场合：

* 系统需要使用现在的类，而此类的接口不符合系统的需要。
* 想要建立一个可以重复使用的类，用语与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。这些源类不一定有很复杂的接口。
* （对对象的Adapter模式而言）在设计里，需要改变多个已有的子类的接口，如果使用类的Adapter模式，就要针对每一个子类做一个Adapter类，而这不太实际。

Factory模式

1. 简单工厂模式，又称静态工厂模式



1. 工厂方法模式



3. 抽象工厂模式

抽象工厂模式与工厂方法模式的最大区别在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。



Singleton模式

要点：

* 类只能有一个实例
* 必须自行创建这个实例
* 必须自行向外界提供这个实例

Strategy模式

Strategy模式就是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。



Strategy模式的实现有以下这些需要注意的地方。

* 经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这些公有的行为放到共有的抽象策略角色Strategy类中。当然这时候抽象策略角色必须要用java抽象类实现，而不能使用java接口。
* Strategy模式在每一个时刻都只能使用一个策略对象，但是有的时候一个应用程序同时与几个策略对象相联系。换言之，在应用程序启动时，所有的策略对象就已经被创立出来，而应用程序可以在几个策略对象之间调换。

在下面的情况下应当考虑使用Strategy模式：

* 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用Strategy模式可以动态地让一个对象在许多行为中选择一种行为。
* 一个系统需要动态地在几种算法中选择一个。
* 一个系统的算法使用的数据不可以让客户端知道。
* 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用Strategy模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现OO设计的概念。

Composite模式

把部分和整体的关系用树结构表示出来。Composite模式使得客户端把一个个单独的成分对象和由他们符合而成的合成对象同等看待。

1. 安全式的Composite模式



1. 透明式的Composite模式



Facade模式

外部与一个子系统的通信必须通过一个统一的门面（Facade）对象进行，这就是Façade模式。

在以下情况下可以使用Facade模式：

* 为一个复杂子系统提供一个简单接口

子系统往往因为不断演化而变得越来越复杂，使用Facade模式可以使得子系统更具有可复用性。

* 子系统的独立性

一般而言，子系统和其他子系统之间、客户端和实现化层之间存在着很大的依赖性。引入Facade模式将一个子系统与它的客户端以及其他的子系统分离，可以提高子系统的独立性和可移植性。

* 层次化结构

在构建一个层次化的系统时，可以使用Facade模式定义系统中每一层的入口。如果层与层之间是相互依赖的，则可以限定它们仅通过Facade模式进行通信，从而简化层与层之间的依赖关系。

Observer模式

Observer模式定义了一种一对多的依赖关系，让多个观察着对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

1．简单的实现方案



2．另一种实现方案

