**第一部分 绪论**

1、在OO开发中，至关重要的能力是熟练地为软件对象分配职责；个人认为将对象进行抽象的能力也相当重要。

2、分析(analysis)强调的是对问题和需求的调查研究，而不是解决方案；设计(design)强调的是满足需求的概念上的解决方案(在软件方面和硬件方面)，而不是其实现。

有益的分析和设计可以概括为：做正确的事（分析）和正确地做事（设计）。

xl

3、需求分析可能包括人们如何使用应用的情节或场景，这些情节或场景可以被编写成用例（use case）。

4、面向对象分析关注从对象的角度创建领域描述，面向对象分析需要鉴别重要的概念、属性和关联。面向对象分析的结果可以表示为领域模型，在领域模型中展示重要的领域概念或对象。

5、面向对象设计关注软件对象的定义-它们的职责和协作。

6、与领域模型表示的是真实世界的类，设计类图表示的是软件类。

7、敏捷建模（agile modeling）强调了UML作为草图的方式，这也是适用UML的普通方式，而且通常对时间投入具有高回报（一般费时较短）。

8、迭代开发（iterative development）是UP和大多数其他现代方法中的关键实践。在这种生命周期方法中，开发被组织成一系列固定的短期（如三个星期）小项目，称为迭代（iteration），每次迭代都产生经过测试、集成并可执行的局部系统。每次迭代都具有各自的需求分析、设计、实现和测试活动。

9、迭代开发的优点包括：

* 减少项目失败可能性，提高生产率，降低缺陷率。
* 在早期缓解高风险（技术、需求、目标、可用性等等）。
* 早期可见的进展。
* 早期反馈、用户参与和调整，会产生更接近涉众真实需求的精化系统。
* 可控复杂性；团队不会被“分析瘫痪”或长期且复杂的步骤所淹没。
* 一次迭代中的经验可以被系统地用于改进开发过程本身，并如此反复进行下去。

10、在复杂、变更系统中（如大多数软件项目），反馈和调整是成功的关键要素。

11、如何进行迭代和进化式分析和设计：

1）在第1次迭代之前，召开第一个时间定量的需求工作会议，例如确切地定义为两天时间，业务和开发人员（包括首席架构师）需要出席。

* 在第一天上午，进行高阶需求分析，例如仅仅确定用例和特性的名称，以及关键的非功能性需求。这种分析不可能是完美的。
* 通过咨询首席架构师和业务人员，从高阶列表中选择10%列表项，这些项目具备以下三种性质：1，具有重要的架构意义；2，具有高业务价值；3，具有高风险。
* 在剩下的一天半内，对这三个用例的功能和非功能性需求进行详细的分析。完成这一过程后，对10%进行了深入分析，90%进行了高阶分析。

2）在第1次迭代之前，召开迭代计划会议，选择上述三种用例的子集，在特定时间内（例如，四周的时间定量迭代）进行设计、构造和测试。

3）在三到四周内完成第1次迭代（选择时间定量，并严格遵守时间）：

* 在开始的两天内，开发者和其他成员分组进行建模和设计工作，在首席架构师的带领和指导下，于“公共作战室”的众多白板上，画出UML的草图（及其他的模型）。
* 然后，开发者摘掉其“建模帽子”并戴上“编程帽子”，开始编程、测试和集成工作并且剩余的时间均用于完成这项工作。
* 进行大量的测试，包括单元测试、验收测试、负载测试和可用性测试等。
* 在结束前的一周，检查是否能够完成初始的迭代目标；如果不能，则缩小迭代的范围，将次要目标置回任务列表中。
* 在最后一周的星期二，冻结代码。必须检入、集成和测试所有代码，以建立迭代的基线。
* 在星期三的上午，向外部涉众演示此局部系统，展示早期可视进展，同时要求反馈。

4）在第1次迭代即将结束时（如最后一周的星期三和星期四），召开第二次需求工作会，对上一次会议的所有材料进行复查和精化，然后选择具有重要架构意义和高业务价值的另外10%到15%的用例，用一到两天对其进行详细分析。

5）于周五上午，举行下一迭代的迭代计划会议。

6）以相同步骤进行第2次迭代。

7）反复进行四次迭代和五次需求工作会，这样在第4次迭代结束时，可能已经详细记录了大约80%-90%的需求。

8）我们大概推进了整个项目过程的20%。此时，可以估计这些精化的、高质量的需求所需工作量和时间。因为具有依据现实得出的调查、反馈结论并进行了早期编程和测试，因此估计能够做什么和需要多长时间的结果会更为可靠。

9）此后，一般不需要再召开需求工作会；需求已经稳定了（尽管需求永远不会被冻结）。接下来是一系列为期三周的迭代，在最后一个周五召开的迭代计划会上选择适宜的下一步工作，每次迭代都要反复询问：“就我们现在所知，下一个三周应该完成的、最关键的技术和业务特性是什么？”

利用这种方式，经过早期探索式开发的几次迭代之后，团队将能够更准确地回答“什么、多少、何时”。

12、建模（构建UML草图……）的目的主要是为理解，而非文档。

**第二部分 初始阶段**

1、用一句话来概括初始阶段：预见项目的范围、设想和业务案例。用一句话来概括初始阶段要解决的主要问题：涉众是否就项目设想基本达成一致，项目是否值得继续进行认真研究。

2、需求分析的最大挑战是寻找、沟通和记住（通常是指记录）什么是真正需要的，并能够清楚地讲解给客户和开发团队的成员。

3、在统一过程中，需求按照“FURPS+”模型进行分类，这是一种有效的记忆方法，其含义如下：

* 功能性（Funcational）：特性、功能、安全性。
* 可用性（Usability）：人性化因素、帮助、文档。
* 可靠性（Reliability）：故障频率、可恢复性、可预测性。
* 性能（Performance）：响应时间、吞吐量、准确性、有效性、资源利用率。
* 可支持性（Supportability）：适用性、可维护性、国际化、可配制性。

”FURPS+“中的”+“是指一些辅助性的和次要的因素，比如：

* 实现（Implementation）：资源闲置、语言和工具、硬件等。
* 接口（Interface）：强加于外部系统接口之上的约束。
* 操作（Operation）：对其操作设置的系统管理。
* 包装（Packaging）：例如无力的包装盒。
* 授权（Legal）：许可证或其他方式。

*若想从生活中得到什么，必不可少的第一步就是：决定想要什么。 -本。斯坦(Ben Stein)*

4、用例是文本形式的情节描述，广泛应用于需求的发现和记录工作中。

5、用例是一种优秀的方法，使领域专家或需求提供者自己编写（或参与编写）用例成为可能，并使这项工作难度降低。

**第三部分 细化迭代1 基础**

1、细化（elaboration）是一般项目中最初的一系列迭代，其中包括：

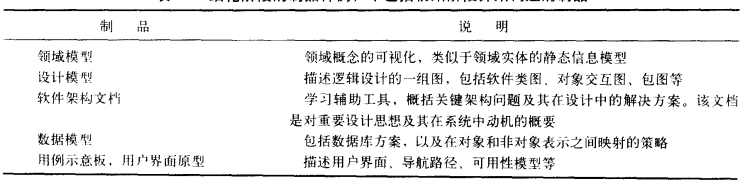
* 对核心、有风险的软件架构进行编程和测试。
* 发现并稳定需求的主体部分。
* 规避主要风险。

细化阶段是最初的一系列迭代，在这一阶段，小组进行细致的调查、实现（编程和测试）核心架构、澄清大多数需求和应对高风险问题。

2、在细化阶段可能出现的一些关键思想和最佳实践包括：

* 实行短时间定量、风险驱动的迭代。
* 及早开始编程。
* 对架构的核心和风险部分进行适应性的设计、实现和测试。
* 尽早、频繁、实际地测试。
* 基于来自测试、用户、开发者的反馈进行调整。
* 通过一系列讨论会，详细编写大部分用例和其他需求，每个细化迭代举行一次。

细化阶段可能构建的制品：



上图并未包含初始阶段就开始构建的制品，而只是细化阶段开始的制品，这些制品不是在一次迭代完成，而是会跨越若干次迭代进行精化。

3、通过关联而不是属性来表示概念类之间的关系。

4、没有所谓唯一正确的领域模型。所有模型都是对我们试图要理解的领域的近似。领域模型主要是在特定群体中用于理解和沟通的工具。

5、在同一层内的对象在职责上应该具有紧密关联，不同层中对象的职责则不应该混淆。

6、需要哪些对象，它们如何通过消息和方法进行协作，通过动态对象建模（例如绘制顺序图）才能真正落实这些准确和详细的结论。

7、应该把时间花费在交互图（顺序图或通信图），而不仅是类图上。

8、任何顺序图都可以使用sd图框围绕起来，并对其命名。当你想要引用相应名字的sd图框时，可以使用ref图框。

9、在类图中，使用依赖线描述对象之间的全局变量、参数变量、局部变量和静态方法（对其他类的静态方法加以调用）的依赖。

10、GRASP是通用职责分配软件模式（General Responsibility Assignment Software Patterns）的缩写。GRASP的9个模式如下所示：

**创建者（Creator）**

如果以下的条件之一（越多越好）为真时，将创建类A实例的职责分配给类B：

* B“包含”或组成聚集A。
* B记录A。
* B直接使用A。
* B具有A的初始化数据，并且在创建A时会将这些数据传递给A。因此对于A的创建而言，B是专家。
* B是对象A的创建者。

注意：对象的创建常常具有相当的复杂性，这时最好的方法是把创建职责委派给称为具体工厂（Concrete Factory）或抽象工厂（Abstract Factory）的辅助类，而不是使用创建者模式所建议的类。

**信息专家（Information Expert）**

信息专家模式是面向对象最基本的原则....

当你设计的一个类,某方面具有完整的信息,(包括完整的知道责任,和行为责任),足以实现某个责任..我们就将这个责任交给这个类..这个类就是所谓的信息专家...

香水也忧伤解析：所谓责任就是一项工作

注意：由于耦合与内聚问题导致某些情况下，专家模式建议的方案并不合适。

**低耦合（Low Coupling）**

分配职责，使耦合性尽可能低。利用这一原则来评估可选方案。

**控制器（Controller）**

控制器是UI层之上的第一个对象，它负责接收和处理系统操作消息。

**高内聚（High Cohesion）**

内聚（或更为专业地说，是功能内聚）是对元素职责的相关性和集中度的度量。

**多态性（Polymorphism）**

 Polymorphism模式提倡通过多态操作把基于类型的可变行为的定义职责分配给行为发生的类。Polymorphism模式把各变化的“行为”定义职责分别分配给具有相同操作行为界面的通用接口的实现子类，利用多态性适应行为的可变性。

**纯虚构（Pure Fabrication）**

Pure Fabrication模式提倡把那些非问题领域的职责分配给那些人工生成的或者容易此类职责的概念类。我们设计对象的时候应该尽量保持与现实世界里的对象一致。这种与现实世界里的对象保持一致的从业务分析中抽象出来的类叫做“Domain Class”。它相当于上述问题领域里的类。一般来说，Domain Class与非Domain Class的功能如果聚集在一个类里，就破坏了“高内聚”原则。

**间接性（Indirection）**

间接性模式关注这样一个问题：为了避免两个或多个事务之间直接耦合，应该如何分配职责？如何使对象解耦合，以支持低耦合并提高复用性潜力？

间接性模式对此的回答是：将职责分配给中介对象，使其作为其他构件或服务之间的媒介，以避免它们之间的直接耦合。中介则实现了其他构件之间的间接性。

间接性模式的思想比较简单，即通过一个中介就能消除许多的耦合。在GoF的23种设计模式中，有许多模式都利用到了间接性的思想。比如桥接模式中，设计将抽象部分与其实现部分相分离，利用的就是在客户与实现之间增加了一个抽象层次。外观模式则是在整个子系统与客户之间增加了一个便于用户使用的外观类作为中介。而中介者模式中的中介者则更是典型的例子。

**防止变异（Protected Variations）**

Protected Variations模式提倡在可预测的变化或不安定因素的周围，用稳定的接口来承担职责。依赖于稳定的接口，将经常变化的代码提炼出一个稳定的接口，让其他的类依赖这个稳定的接口，变化的部分继承这个接口后一般在子类中实现。