# 第5章 分析阶段

通过需求分析阶段，我们以用户可理解的方式表达了需求。然而，软件设计是一个技术性、专业性的工作，用户的需求和软件设计之间存在较大的“知识鸿沟”和“表示鸿沟”。通过分析阶段，开发者对需求有了自己的理解，一方面对需求的正确性、全面性、可行性进行检验，另一方面，建立需求与软件设计之间的桥梁，从而使得后续阶段中软件开发工作能够顺利开展。因此，分析阶段依旧是围绕需求展开的，并不涉及到软件设计的细节，但是由于采用了开发者的观点来分析需求，因此，必然会引入一些与软件相关的概念。例如，如果软件开发采用了面向对象方法学，在分析阶段将会获取对象模型，并通过动态模型来描述对象之间是如何通过交互来提供功能的。对象模型，动态模型这些都是软件开发相关的概念，但是，在分析阶段中，涉及到的类（对象）并非是软件类（对象），而是“领域类（对象）”，即对应于用户应用中概念的类（对象）。在后续设计阶段，“领域类（对象）”将通过对应的软件类（对象）加以具体实现，分析阶段就是以这种方式建立了需求与设计之间的桥梁。通过分析阶段，需求获取阶段的结果在被检验后可能会被修改。

## 分析阶段的主要内容

分析阶段与三个模型相关，即功能模型、对象模型和动态模型。在需求获取阶段得到的功能模型是分析阶段的输入，通过分析，建立系统的对象模型和动态模型是分析阶段的输出。其中，对象模型以面向对象的方式定义了系统的内部构成，因而主要以类图的方式进行表达。动态模型表达了围绕用例而展开的对象之间的动态交互过程、某些对象的复杂状态变化以及与软件相关的复杂的业务过程，动态模型可以用交互图（顺序图或者协同图）、状态图或者活动图进行表达。在构造对象模型和动态模型的过程中，可能会对功能模型进行修改。

分析阶段是一个迭代的过程，它涉及到以下的一些步骤：

1. 类的识别

在采用面向对象方法学进行软件开发时，软件的构成单元是类以及它们的对象，因此，确定正确的类（对象），就是确定软件的构成。在分析阶段，确定类的依据是需求和领域知识。值得指出的是，这些类并非是软件概念上的类，它们是领域类，通过设计阶段的工作后，领域类将与软件类相对应，换言之，通过软件类来实现领域类。

1. 对象交互描述

围绕每个用例，各个类的对象之间需要进行交互，才能完成用例描述的流程。对象交互的描述在分析与设计中都具有极其重要的意义。在分析阶段，对象交互的描述规定了每个类具有的责任，并形成了对象之间的协作关系；而在设计阶段对对象交互的描述，确定了每个软件类必须提供的操作（需要处理的消息）。不同的交互方式将最终对系统的可维护性、可扩展性、性能等产生决定性影响。同时，通过交互方式的描述也可以进一步发现前期需求中的不足，以及发现更多的相关的类（对象）。

1. 类行为刻画

部分类可能具备复杂的行为，特别是其行为受到其状态的影响。在这种情况下，我们需要描述其可能的状态，以及状态之间的转换条件。我们通常可以采用状态图来对类的行为进行刻画。当然，那些不具备复杂的状态的类并不需要构造状态图。

1. 对象模型构建

在识别类（对象）、描述对象交互行为的基础上，我们可以形成完整的对象模型。各个类的属性、操作可以通过前述的分析得到。同时，在对象模型中，类之间的关系也将得到表达，包括关联和泛化关系。

1. 迭代和检查

以上四个步骤不是一个线性的过程。实际上，类的识别、对象交互描述和行为刻画、对象模型构建构成一个相互影响的迭代序列：识别出来的类的对象将参与交互过程，在描述交互过程中，会引入新的类（对象）。通过多轮迭代后，最终得到的模型必须是正确的、完整的、一致的和现实的。所谓正确的，指的是模型确实代表了实际的需求；模型是完整的，意思是每一个场景包括意外场景都得到了描述；模型的一致性指的是模型中的元素互相不冲突；模型是现实的，意味着模型是可以得到实现的。

分析阶段得到的模型依旧表述在软件需求规格说明文档中，并且主要是对系统模型这一章进行修改和细化。具体来说，对用例模型这一小节，可能会带来修改，同时补充对象模型和动态模型这两小节的内容。

## 对象模型的创建

### 类的识别

分而治之是人类处理复杂性的重要手段，因而这也是软件开发中采用的重要手段。不同的方法学采用了不同的分解手段。传统的结构化方法是将系统按照功能进行模块分解，而面向对象方法学是按照类（对象）进行系统分解。因此，类的识别在面向对象软件开发中是一个基本任务。在实践中，许多开发人员从自己的偏好和经验出发进行对象识别，采用的方法过于随意，这将大大降低面向对象方法学的好处。

1. **类的识别方法**

按照面向对象方法学的理念，我们应该从实际问题中去找类，这种找到的类我们称为问题领域类，它主要是从实际问题出发，反映了实际问题中的业务实体、业务过程和业务概念。类的寻找有以下方法：

（1） 文本分析方法：可以从用例文档中、特别是词汇表中找出那些意义重要的一般名词，它们是潜在的类。

（2） 重用已有类概念：如果在该领域已经有一些软件，可以沿用该领域经常采用的类的概念。

1. 从动态模型中识别：在构建动态模型的过程中，可能会添加新的类。
2. **类的类型**

在识别类时，我们可以将类分为三种类型，分别为实体类，边界类和控制类。实体类代表了系统中需要跟踪的持久化信息实体，它是业务实体的直接代表；边界类负责用户与系统之间的交互，例如系统窗口类；控制类代表了系统中的控制功能，它负责协调多个对象完成用例流程。三种类别的UML表示方式如图5-1所示。



图5-1 实体类、边界类和控制类的表示

将类分为这三类体现了软件工程中“视图-模型”相分离的原则，实体类对应于“模型”，边界类对应于“视图”。模型对应于业务对象，是相对稳定的，而视图则可能变化比较多，通过分离，有效地降低了软件维护的成本。类的这三种类型也可以类比一个企业内的组织分工，实体类就是那些岗位员工，是真正处理业务的；边界类就是那些销售和供应人员，负责与外界打交道；控制类是企业中的管理人员，本身不干活，负责协调大家。一个合理的组织分工这三者都是需要的。

1. **类识别注意点**

在识别类的时候，有以下注意点：

1. 类的命名：在类命名时，首先应该用实际业务中的名字，同时，名字应该相对通用，而不是过早限定具体的方式。例如，在教室预定系统中，代表终端的类，可以是MobileTerminal，也可以是MobilePhone，MobileTerminal不限于MobilePhone，是不是客户端只支持MobilePhone可以留待后续设计时决定，那么用MobileTerminal就会比较通用。
2. 虽然我们一再强调类应该代表现实中的概念，然而我们毕竟在开发软件，所以在识别类时，可能会专门创造一些用来记忆“公共信息”的类，以避免信息的重复。例如，在现实生活中，每一件商品上都是有商品信息（名称、产地、成分等等）的，但是同类商品上的这些商品信息都是一样的，为了避免重复，我们可以添加一个ProductSpecificaion类，记录商品信息，Product类本身只保存商品编号、生产日期等信息，从而避免信息的冗余。
3. 在识别类时，也有可能依据软件的具体需要，适当添加一些实际生活中并不存在、但是出于软件的特点而添加的类，控制类就属于这种情景。

### 对象模型的表达

识别出类后，我们将利用类图来表示对象模型。类图由类和类之间的关联关系构成。分析阶段的对象模型主要是对真实世界中概念类的表示，而不是软件对象的表示。对象模型中要考虑：

1. **属性的添加**

每一个类需要添加属性，类的对象的属性取值保存和刻画了对象当前的状态。每一个类都有许多细节信息，因此可以作为属性的数据项也很多，但是，我们并不需要也不可能把一切数据项都作为属性。我们可以根据该属性是否对我们当前的应用有价值来确定是否需要添加该属性。

另外一个问题是哪些应该作为属性，哪些作为类来建模。一般而言，属性应该是基本数据类型（如整形、浮点型、布尔型、字符串型）。如果一个属性是复杂数据类型（可以进一步分成多个数据项的数据），最好把它作为单独的类，然后将这两个类关联起来。

1. **关联的添加**

关联刻画了两个类结构方面的联系。那么，我们需要把所有可能的关联关系都刻画出来吗？显然并不必要。因为从某种意义上讲，世界上的万事万物都是关联的，小世界定理甚至告诉我们世界上任意两个人只要通过不多于六个中间人就可以联系起来。那么我们需要添加什么样的关联关系呢？以下为一些指南：

1. 添加那些重要的、需要持续一段时间的关联信息；
2. 添加那些能够表示聚合、组合这种“整体-部分”关系的关联信息；
3. 在此阶段，我们只要从领域知识出发来考虑关联，并不需要考虑关联将来如何在软件中实现的问题；
4. 不要添加那些可以从其他关系中推导出来的关联关系。

完整的关联信息包括了：关联的方向、关联的名字、多重性、角色。

1. **对象模型的精化**

对象模型的构建并不是一次完成的。我们应该对建立的对象模型进行多次重新检查并在必要的时候进行模型的调整。

在模型中可能会发现如下问题：

1. 重复的定义：由于我们通过多种渠道（如通过文本分析、构建交互图、依据领域知识等）去识别类，因此有时候可能将一个类重复定义了两次，或者将同一属性、同一操作定义了两次，我们需要将它们进行合并；
2. 应该建模为类的属性，或者不必要作为类的属性：有些属性类型比较复杂，或者有相应的操作与其关联，那么最好将其建模为类，而相反，有些类只有很少的属性，并且只有写入和读取的操作，那么就可以简化其为属性。

随着分析的深入，可能会发现以下情况从而可以建立类之间的“泛化-例化”结构：

1. 一个概念在现实中可以分成子类型，并且子类型具有不同的行为，那么可以将子类型建模成子类。但是值得注意的是，并不是在现实中的子类型都需要建模成子类，过于复杂的继承结构也会造成软件的复杂，因此只有在子类型对于应用有意义时，才需要建模成子类。
2. 发现多个类具有共同点，因此抽象出一个父类。通过抽象出的父类，并定义继承结构，可以增加重用，改善软件的可维护性。

判断两个类之间是否存在泛化-例化关系时，有以下方法：

1. 百分百法则：父类的属性、方法和与其他类的关系可以百分百运用到子类上；
2. IS-A法则：如果ClassA是ClassB的父类，那么“ClassB的对象IS-A ClassA”这句话就应该能读通。

## 动态模型的创建

对象模型刻画了系统的静态构成，显然，我们还需要表示系统各组成部分之间的动态关系，这种动态关系我们用动态模型进行表示。创建动态模型的依据依旧是用例模型，其基本任务就是用一系列UML中的模型图将用例中的文字描述进行图示化的表达。

动态模型中可以包括下列具体的模型图：

* 交互图：构造系统顺序图，识别系统消息，围绕每一条系统消息，进行交互图的构造。
* 状态图：针对具有复杂状态、特别是在不同状态下，将呈现出不同行为的类构造状态图。
* 活动图：我们可以针对一个用例，或者一个业务流程用活动图进一步进行刻画。

### 交互图的构建

尽管初学者认为对象模型对软件的实现具有直接的指导意义，但是从软件开发的角度来看，交互图是真正体现设计思想的地方，或者说，它是一个根据用例来定义对象模型的工具。

我们给出交互图的步骤：

1. 构造系统顺序图；
2. 选择部分系统消息编写操作契约；
3. 针对每一个系统消息，构造交互图。

#### 系统顺序图

为了更好地刻画用户与系统的交互，我们针对每一个用例，将参与者与系统之间的交互进行表达，参与者给系统发送的消息我们称之为系统消息。显然，我们要开发的系统必须处理每一条系统消息。因此，通过画系统顺序图能够为交互图的构造提供准备。

在系统顺序图中，一端为发起用例的参与者对象，一端为系统。我们将系统作为黑盒子。如果涉及到与此系统交互的其他系统，也可以将它加入其中。



图5-2 系统顺序图的例子

系统消息的命名一般表达为一个动宾词组。

#### 编写系统操作契约

对于一些复杂的系统消息（对应于系统操作），我们可以进一步采用系统操作契约（System Operation Contract）定义其对对象模型中的各种对象产生的影响，这些影响将来就会在交互图中加以具体刻画和体现。

操作契约包含以下内容：

1. 操作名称：给出操作名称及参数；
2. 交叉参考：给出该操作属于哪一个用例；
3. 前提条件：在执行该操作前，系统或者对象模型中涉及的对象应该满足的条件；
4. 后置条件：该部分最为重要。用来记录执行完该操作后，系统或者对象模型中的对象将会发生的变化。

后置条件并非用以说明执行该操作中要完成的任务，而是记录了该操作执行完毕后，系统中发生的各种变化。这种变化可以分为三种类型：

* 引起了对象的创建或者删除；
* 引起了某些对象内部属性值得改变；
* 在某些对象之间形成了关联关系或者原有的关联关系消除了。

以申请使用教室用例为例，该用例中的一个系统操作为**提交申请（submitApplication），**契约描述如下：

**契约CO1：submitApplication**

|  |
| --- |
| * **操作：**submitApplication(applier:Applier, applyDate:Date, applyTime:Time, applyPurpose:String, numberOfUsers:integer, classroomID:ClassroomID) * **交叉引用：**申请使用教室 * **前置条件：**正在进行的教室使用申请 * **后置条件：**   + 创建了ClassroomApplication的实例ca   + 基于applier.applierID的匹配，将ca与Teacher或Student关联   + ca.applyDate赋值为applyDate   + ca.applyTime赋值为applyTime   + ca.applyPurpose赋值为applyPurpose   + ca.numberOfUsers赋值为numberOfUsers   + 基于classroomID的匹配，将Classroom关联到ca |

定义操作契约带来的好处就是为我们构造交互图给以了明确指导。

需要再次指出的是，并非每一个系统消息都需要定义系统操作契约，我们只需要为那些可能引起比较复杂的变化的消息定义操作契约。

#### 交互图构造

交互图可以分成顺序图和协同图，顺序图与协同图在语意上是相等的，画一个图就可以生成另一个图，因此我们将以顺序图为例来讲解交互图的构造。

1. **软件工程的原则与责任分配**

在画顺序图时，从一个对象（对象A）发送一个消息给另一个对象（对象B），其意义在于对象A请求对象B的服务，换言之，将某一个责任分配给了对象B。那么，为什么将责任分配给对象B而不是其他对象呢？这后面的依据是软件工程中的一个核心思想：高内聚和低耦合。高内聚和低耦合的提出能够提高软件的可理解性和可维护性。

类的高内聚有两层意思：

* 一个类的功能应该比较单一，这可以称之为语意内聚；
* 类内部的各个方法调用可以比较多，这称之为关系内聚。

低耦合的意思是各个类之间的关系应该尽可能少。

显然高内聚、低耦合使得某一个类的修改对于别的类的影响会比较小，同时，系统的重用性也会比较高，因为重用系统的某一个部分不会让其它的类过多地牵涉其中。

如何进行设计才使得系统能够高内聚、低耦合呢，在分配每一个责任时，我们都必须考虑这个原则。体现这一原则的一个做法是，每次我们分配责任时，我们先去看哪个对象拥有完成该责任所需要的信息，谁拥有信息，谁就承担这个责任。这显然与我们的生活常识相匹配，在办一件事情时，我们得去找拥有信息干这件事情的人，这样最为直接，而不是绕几个圈子找其他人办事使得事情更为复杂。

高内聚、低耦合的原则在使用时也并不是金科玉律，在某些时候，与一些类库、提供公共服务的类如数据存储服务类的耦合是可以接受的，因为它们本身是比较稳定的。

上述谈及的高内聚、低耦合原则似乎是针对软件设计的。我们前面又多次声明分析阶段还是侧重于理解需求，因此分析阶段获取的类并非软件类，那么为什么我们也强调需要运用这个原则呢？因为此处分析出来的对象模型、动态模型是后续设计的直接指导，在设计阶段我们实际上是在对象模型、动态模型上进一步细化，并按照软件系统的一些特点做一些调整，因此，在分析阶段我们就需要贯彻此思想。

1. **顺序图的构建**

我们围绕每一个由参与者发出的系统消息都需要单独构造一张顺序图。当然，如果围绕这一系统消息的顺序图比较简单，可以和相关的系统消息的顺序图合并。在画顺序图时，可以遵从以下的规则：

1. 左边第一列为发出系统消息的参与者对象；
2. 第二列一般为一个边界类对象，代表与用户交互的界面；
3. 第三列一般为控制类对象，由边界类对象创建，该对象负责协调接下来的流程。通常有两种控制类对象，一种为针对一个用例的控制类对象，它负责这个用例的所有流程协调；一种为针对一个子系统或者整个系统的控制类对象，这种情况下子系统或者整个系统中用例较少，因而不再单独为每个用例去创建一个控制类对象。用例控制类对象的生命周期一般与用例一样，即用例流程完成后，用例控制类对象就结束期生命期。
4. 第四列可能是控制类对象创建的一些其他边界类对象，或者是实体类对象；
5. 其他不是发出系统小的参与者一般画在最右边。

下图为一个顺序图的例子，针对的是系统消息SubmitApplication().



图 5-3 顺序图的例子（申请者提交申请且成功时的顺序图）

### 状态图

状态图是对具有复杂状态的类的进一步刻画，并非每一个类都需要构造状态图。我们需要注意状态图与顺序图的差别。状态图针对的是单个类，刻画了类的对象在整个生命期内可能历经的状态变化。顺序图针对的是多个类的对象，它们协作完成整个用例流程。

教室的状态图的例子如下。



图 5-4 状态图的例子

## 软件需求规格说明的修改

分析阶段的结果依旧表述在***软件需求规格说明***文档中，主要体现在对“系统模型”这一节的细化上。词汇表由于新识别出一些类，可能需要进一步补充。软件需求规格说明文档的其他部分也可能需要修改。

1. 场景

如果有补充的场景，在此处添加。

1. 用例模型

如果有补充和修改的用例，在此处添加。

1. 对象模型

分类别列出每一个类的介绍

表 实体类定义表

|  |  |  |  |
| --- | --- | --- | --- |
| 实体类名称 | 属性 | 关联类 | 定义 |
|  |  |  |  |
|  |  |  |  |

表 边界类定义

|  |  |
| --- | --- |
| 边界类名称 | 定义 |
|  |  |
|  |  |

表 控制类定义

|  |  |
| --- | --- |
| 控制类名称 | 定义 |
|  |  |
|  |  |

列表方式介绍完类后，画出一张类图。

1. **动态模型**
2. 系统顺序图与操作契约
3. 用例X

* 为每一个用例画出一张系统顺序图
* 为该用例中的系统消息定义操作契约

1. 顺序图

围绕上述系统顺序图，为其中每一个系统消息单独画一张顺序图。。

1. 状态图

给出某些具有复杂状态的类的状态图

1. **用户界面**

在需求获取阶段的基础上，进一步细化界面。