

《软件方法》2026版



含 1、2、8、9 章部分内容

本文件会不定时更新，可到以下地址下载最新版本：

umlchina.com/url/softmeth.html

当前更新时间：2026.6.5

您在阅读《软件方法》时如果发现错误，欢迎通过微信[umlchina](#)告知。如果作者认为有道理，决定在下一次发布时根据您的意见修改，每个错误将付给您5.12元报酬，并在书中说明您的贡献。

- (1) 任何您认为的错误都可以，包括错别字。
- (2) 同一错误仅支付最先指正者报酬。
- (3) 请根据最新版本作指正。

目前指正人有（按指正时间排序）：吴佰钊、王周文、刘学斌、成文华、黄树成、李蜀斌、杨雪鸿、王书伟、高洪江、张志坚、龙燧、陈文飞、郭沼兵、陈自平、张彬、李宏伟、赵志军、孙赛刚、孙军、刘京城、左科、王长福、张守金、杜秀清、侯文娟、顾正海。

2018 版《软件方法（上）》的勘误：

umlchina.com/book/errata2ed.html

第 1 章 ABCD 工作流

牵着你走进傍晚的风里，看见万家灯火下面平凡的秘密。

《情歌唱晚》；词：黄群，曲：黄群，唱：曹崴；1994

1.1 利润 = 需求 - 设计

1.1.1 利润 = 需求 - 设计

利润 = 收入 - 成本。不管出售什么，要获得利润，需要两个条件：

- (1) 售价要高；
- (2) 成本要低。

妙就妙在，价格和成本之间没有固定的计算公式，这正是创新的动力之源。

放到软件业上，我也炮制了一个式子：

$$\text{利润} = \text{需求} - \text{设计}$$

在软件开发中，需求工作致力于解决“提升销售”的问题，设计工作致力于解决“降低成本”的问题，二者不能相互取代。能低成本开发和维护某个系统，不一定能保证它好卖。系统好卖，如果开发和维护成本太高，最终利润还是高不了。

以上说的“利润”、“销售”、“成本”是广义的，可以是金钱，也可以是名声、权力、人力等。

即使你做的是没有在市场上公开出售的单位内部系统，甚至是自己为自己的方便做一个系统，也适用上面的式子。

更为关键的是：需求和设计之间不存在，也不应该存在刻板的一对一映射——也幸亏如此，否则人工智能现在就可以轻易学习其中的映射套路，取代软件开发人员完成所有的工作。目前，软件开发人员存在的价值之一就是根据自己掌握的领域知识和软件开发知识，努力在很多可能的映射方案中挑选出最合适的映射方案，以及创造出更好的映射方案。

我们先来看自古以来就有的一个系统，人，即“人脑系统”。

人脑系统的功能（需求）是（人能够）走路、跑步、跳跃、举重、投掷、游泳……但是设计人脑系统的结构时，并不是从功能（需求）直接映射到设计，得到“走路器官”、“跑步器官”、“跳跃器官”……人脑系统的器官是眼、耳、心、肺、肝、胃、骨架、皮肤……这些器官和人脑系统的功能不是一一对应的，在互相协作以完成系统的功能时，它们和功能之间的关系是多对多的。

图 1-1 表达了某个人脑系统的需求和设计。可以看到，需求和设计的映射是多对多的。各个器官被各个功能共享，不能说“心脏是老板的”、“肺是老婆的”。

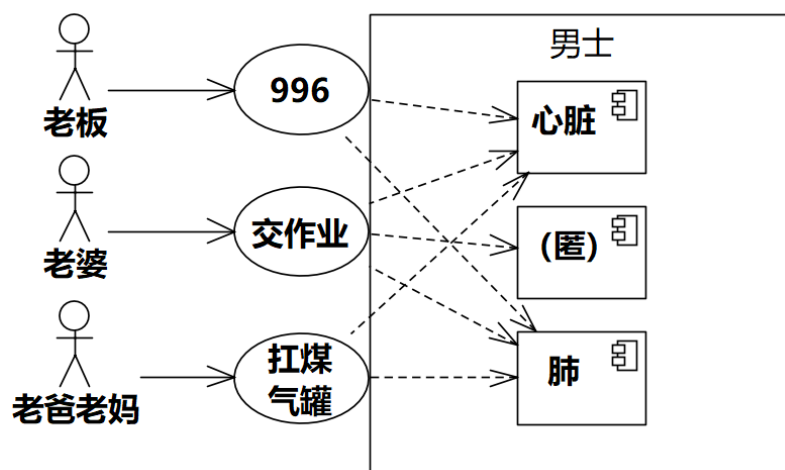


图 1-1 人脑系统的需求和设计

★这里要注意一种无知的论调：干嘛要共享，每个功能给它一个心脏不行吗？真实的心脏是肉做的，复制要花成本，软件的代码复制又不用花钱！

图 1-1 的映射只是造物主——用《异形 (Alien)》的说法就是工程师 (Engineer) ——当初挑选的映射方案。

今天，人类也成为造物主开始“造人”。同样是造能跑能跳能扛的“人”，我们挑选的映射方案可能和人类的造物主 (如果存在) 当初“造人”的映射方案不一样，如图 1-2。当然，目前人类的科学技术水平，连细胞都造不出来，更不用说做到一模一样的方案了。

“工程师”当初制造的人



人类制造的“新人”



图 1-2 人类和“新人类”

“人”的使用者并不关心这个“造人”的方案。

老板要雇民工扛煤气罐，他只要求这个民工能跑能扛，管他体内构造是心肝脾肺肾（如图 1-2 上部）还是电路板（如图 1-2 下部）——如果电路板民工更便宜，他会毫不犹豫地 淘汰掉心肝脾肺肾民工。

民工找工作也要从市场的需要来找——“老板，你要雇人扛煤气罐吗，我可以！”，而不是从自己的内部器官出发来找——“老板，我每天都管理我的心脏，你请我吧！”

以上所说的这些，总体意思就是：要学会把需求和设计分开，这也是贯彻全书的核心思想，后面还会反复强调。当然，用词可能会有变化，可能有的时候会说“卖和做分开”，有的时候会说“外和内分开”等等。

软件开发中，如果从需求直接映射设计，会得到大量的重复代码，成本增加；如果从设计出发定义需求，会得到一堆假的“需求”，销量下降。不管是哪一种情况，利润都会下降。

而这一点，很多软件开发人员并没有意识到。

1.1.2 需求-设计混淆的常见错误

1.1.2.1 “子系统” 其实是需求包

我们经常听到这样的说法，“本系统分为八大子系统，包括销售子系统、财务子系统、库存子系统……”，这就是需求和设计不分的一个例子。其实，说话人可能应该这样表达自己的意思：“本系统的功能需求分为八大需求包……”。

需求包是基于涉众视角对系统功能分包而得到的，子系统(用 UML 的说法是组件)是基于内部视角根据系统部件的耦合和内聚情况切割而得到的，这两者不是一一对应的。所谓的“财务子系统”，其实可能是“把财务人员使用的功能放在一个包里”，如图 1-3。

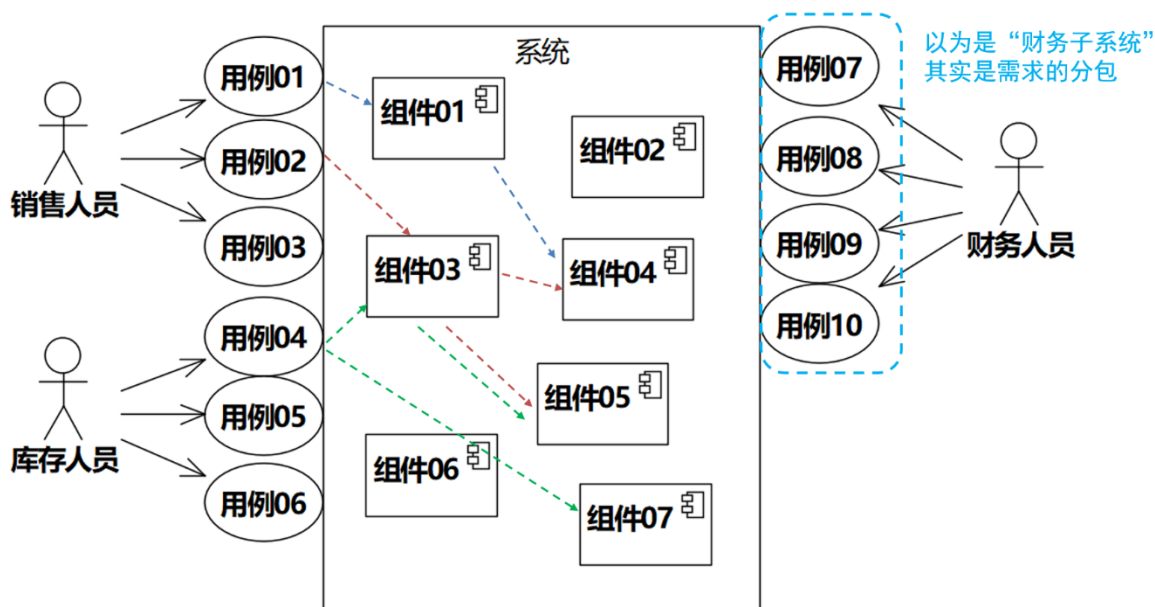


图 1-3 “子系统” 其实是需求的分包

1.1.2.2 “功能模块”是错误用语

另一个常见的需求和设计不分的说法是“功能模块”。

功能 (Function) 这个词，如果应用于信息系统，描述的是：系统作为一个整体，接收到外部请求时应该产生的效果。

模块 (Module) 这个词，如果应用于信息系统，描述的是表示系统分解后得到的各个组成部分。

“功能模块”意味着在意识里认为“功能”和“模块”有直接的映射关系，甚至认为“模块”是属于某个“功能”的模块，是为了完成某个“功能”而存在的。

用前面的“人脑系统”来举例，“功能模块”就是“走路器官”、“走路子系统”……，这是错误的。

不但“功能模块”不应该连起来说，而且“功能”和“模块”这两个词的含义也是模糊的。

例如，以自助柜员机 (ATM) 为目标系统，“取现金”是“功能”，“登录”也是功能，“计算手续费”也是“功能”，到底“功能”有多大？“模块”是一个控件？还是一个类？还是若干个类形成的组件？

因此，本书后文中，会尽量使用更精确的术语。例如，以 ATM 为目标系统，“取现金”是一个用例，“登录”是用例中的一个回合，“计算手续费”是一个步骤，系统里面可能会有一个“账户”类……。

1.1.2.3 微服务的遮羞布

最近几年鼓吹的新词“微服务”造成一定的误导。有的人误以为“微服务”就是“需求设计——对应”。

假设考虑到开发团队的结构，把系统分成多个“微服务”，分由各个小团队应用各自的技术栈独立完成。例如图 1-1 中的男士，可能会被分割为“996 微服务”、“交作业微服务”、“扛煤气罐微服务”。

且不说这样划分是否合理，即使这样划分了，“微服务”内部也要通过自己的各个部件（可能是残缺的）协作完成，例如“交作业微服务”要完成“交作业”用例，也需要眼睛、耳朵、手、脚、心脏、**等（可能是残缺的）协作完成，并非映射一个“交作业模块”然后就搞定。更何况，有的用例需要若干个“微服务”协作才能完成。

另外，“微服务”是妥协的不良结构。如果这样的划分风格所得到的软件结构真的是良好的结构，我们几十年前就可以这样做，难道之前的复杂系统没有分解吗？不需要分解吗？如果这真的是好的结构，放在什么时候都是大杀器，根本不用等到现在，拿“微服务”（这也是一个造词）之类的作为理由。即使是一个人完成的项目，也不妨引入一个假设“由各个小团队应用各自的技术栈独立完成”来改善软件的结构嘛。

“微服务”所要解决的问题，在某些系统中确实是存在的，但我在这里要提醒读者，提防把“微服务”（或类似手段）当成遮羞布。

一个人没有能力解决主要问题时，他可能会采取一些措施来掩盖自己能力不足的事实，其中之一就是引入次要问题。

这也是人之常情了。例如，我们在工作中碰到头痛的问题，有时会逃避着不去碰它，而去做一些整理文件，回复邮件等琐碎的事情来获得成就感。

我用盖大楼作类比：

两座大楼耸立在那里，要判断地震来了哪座大楼不容易塌，要考虑的是大楼的结构、所用的材料、所在位置的地质环境等，与大楼由哪几家工程队建造，分别收了多少钱等等已经没有直接关系，甚至与大楼是猫建造的、狗建造的、外星人建造的，也已经没有直接关系——毕竟大楼已经在那里了。

但要研究这些直接影响因素，涉及到艰深的工程力学、流体力学、岩土力学等知识。架构师李三没把这些知识学扎实，正在那里犯愁呢。

这时，伪创新专家张四出现了。张四说，时代变了，现在盖楼要讲“新建筑学”，要考虑到人际关系，要搞好团结。

于是，李三想着反正“老的”建筑学我也搞不懂，还是搞“人”轻松一些。这样吧，有几个工程队要分润，就分几个包，大家开干就是。

转换思想后，李三每天累并快乐着，和兄弟们灯红酒绿，推杯换盏。而且，运气好的时候，盖出来大楼确实也能住人。

如果李三说，公司又不是我的，我就是个打工的，想那么多干什么？张四老师的“新建筑学”省力省心，面上也过得去，刚好是我想要的。

这可以理解。

如果李三说，老板其实也不关注质量，他要的就是在某某大日子到

来之前有个样子货好交差。张四老师的“新建筑学”在这方面很“实用”。

这也可以理解。

如果李三说，这么干，有利于整个建筑团队的安定团结。

这也可以理解。

如果李三说，张四老师这个“新建筑学”盖出来的大楼更抗震，他应该到清华大学建筑学院开一门课“划时代革命性建筑学”，取代“老的”建筑学。

这是不能容忍的！

1.1.3 总结

我简要归纳需求和设计的区别如图 1-4, 在后面的章节中再慢慢进一步阐述这些区别。

需求	设计
卖的视角	做的视角
具体	抽象
产品当项目做	项目当产品做

图 1-4 需求和设计的区别

★高焕堂在他的书《USE CASE 入门与实例》[高 2008]中说过：用例是收益面，对象是成本面。本书基于他的思想做了扩展。

1.1.4 自测题



本书不提供习题答案，请扫码或访问 umlchina.com/book/quiz1_1.html 自测，做到全对自然就知道答案了。



1[单选]

软件开发中需求工作的目的是_____。

- ☐A) 让系统更加好卖
- ☐B) 更好地指导设计
- ☐C) 对系统做概要的描述
- ☐D) 满足软件工程需求规范

2[单选]

软件开发中设计工作的目的是_____。

- ☐A) 对系统做详细的描述
- ☐B) 更好地指导编码

- ☐C) 降低开发维护成本
- ☐D) 满足软件工程设计规范

3[单选]

开发人员说“根据客户的需求，我们的系统分为销售子系统、库存子系统、财务子系统.....”，这句话反映了开发人员可能有什么样的认识错误？

- ☐A) 开发人员没有认识到面向对象设计的重要性
- ☐B) 开发人员直接从设计映射需求
- ☐C) 开发人员直接从需求映射设计
- ☐D) 开发人员没有用 UML 模型来描述子系统

4[单选]

打开开发人员写的需求规约，发现用例的名字都是“学生管理”、“题库管理”、“课程管理”.....，这背后可能隐藏的最大问题是什么？

- ☐A) 用例的名字不是动宾结构，应改为“管理学生”.....
- ☐B) 用例粒度太粗，每一个应该拆解成四个用例，“新增学生”、“修改学生”.....
- ☐C) 开发人员直接从需求映射设计
- ☐D) 开发人员直接从设计映射需求

5[单选]

以下这些经常在开发团队里使用的词汇，都是不严谨的。其中_____混淆了需求和设计的区别。

- ☐A) 功能模块
- ☐B) 详细设计
- ☐C) 用户需求
- ☐D) 业务架构

6[单选]

关于需求和设计，以下说法正确的是_____。

- ☐A) 需求关注概要、设计关注详细
- ☐B) 需求的目的是更好地指导设计
- ☐C) 设计的目的是把系统分解成可以编码的模块
- ☐D) 需求和设计不是一一对应的

7[多选]

有的开发人员没有能力剖析复杂领域逻辑，于是引入性能、开发时间、开发团队组成等遮羞布来转移问题的焦点。以下的哪些做法和这样的做法类似？

- ☐A) 医生在给癌症患者看病时，偷偷借助人工智能来帮忙。
- ☐B) 医生在看病时，对癌症患者说“你经济条件不好，目前只能吃得起这个药，所以这个药最合适”。

☐C) 医生在看病时，对癌症患者说“我们医院目前只有这个药，所以这个药最合适”。

☐D) 医生看到癌症患者经济条件不好，于是 80000 元一盒的药只收 800 元。

1.2 建模 workflow

1.2.1 建模 workflow ABCD

要做好需求和设计，从而低成本制造出好卖的系统，并非喊喊口号就可以，需要静下心来学习和实践一些必要的建模技能。

软件开发是增量、迭代进行的，每一个迭代周期都需要依次思考这么几个事情：

A-业务建模 (business modeling) ——定位需要改进的目标组织以及该组织接下来最需要改进的问题。

B-需求 (requirements) ——描述为了改进组织的问题，待引入的信息系统必须具有的整体表现。

C-分析 (analysis) ——提炼为了满足功能需求，待引入的信息系统需要封装的核心域机制。

D-设计 (design) ——考虑质量需求和设计约束，将核心域机制映射到选定非核心域上实现。

本书将以上 ABCD 称为软件开发的 4 个建模 workflow。

如果采用本书的方法学以及 UML 表示法,其 ABCD 大致如图 1-5。
读者先看一眼即可, 后文再详细解说图中内容。

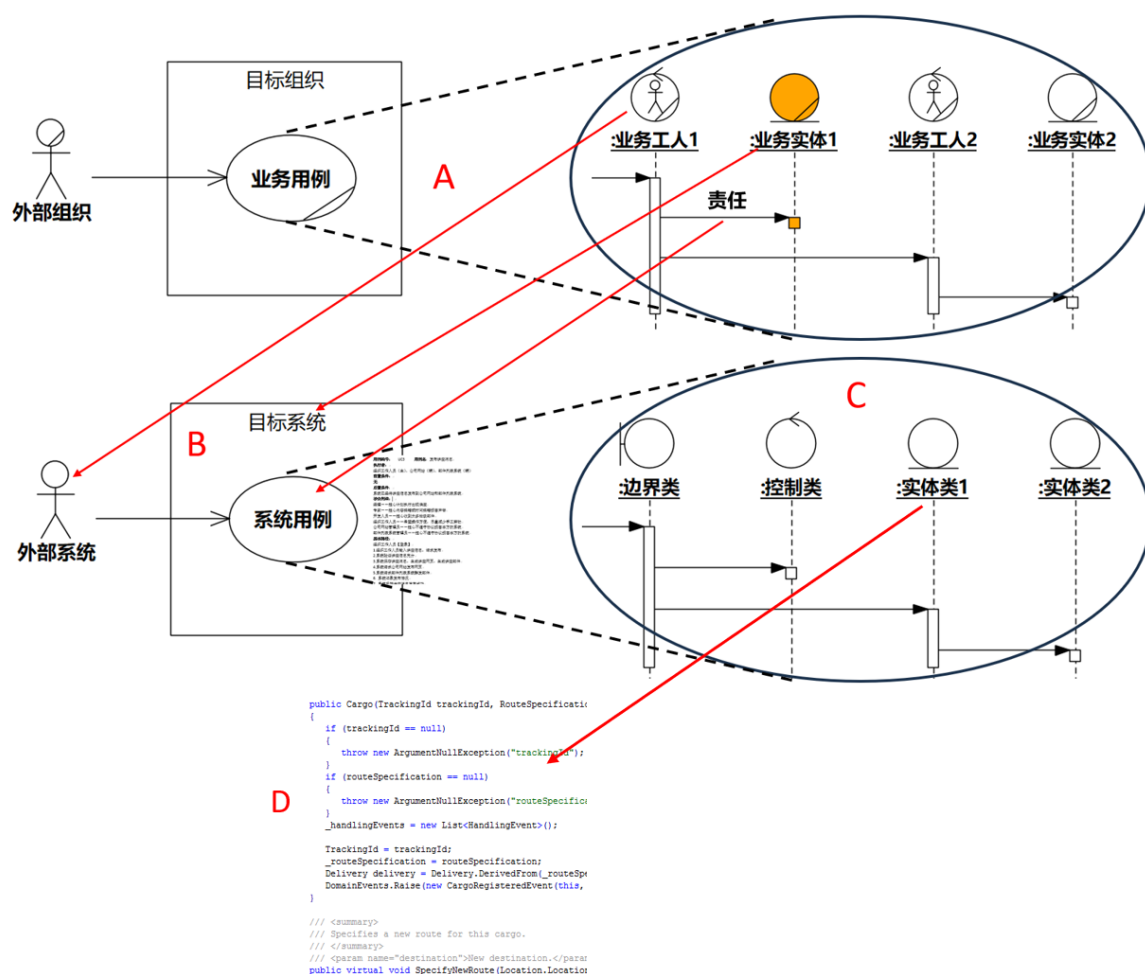


图 1-5 四个建模 workflows (本书方法学+UML)

软件江湖中各种花里胡哨的用语, 不管是真创新还是伪创新, 基本上都可以用上面的 ABCD 来归纳, 例如:

产品经理、需求工程师、需求分析师: A+B+部分 C。

业务架构师: 可能是 A (此时“业务”的含义是“组织”), 也可能是 C (此时“业务”的含义是“核心域”)。

系统架构师: C+D。常有团队说要改进系统架构, 其实他想改进的是 B-需求。

领域驱动设计：C+D。也有团队声称要学习“领域驱动设计”，其实想解决的却是 A-业务建模的问题。

中台：C+D

微服务：C+D

设计模式：C+D

.....

欢迎读者随时补充更新。

1.2.2 关于用词

“ workflow (workflow)”沿用自 RUP (Rational Unified Process) 的早期版本，后来的 RUP 版本改为“科目 (discipline)”。

在我看来，“ workflow”和“科目”这两个词都不算太好，但目前没有更合适的选择，暂时先用“ workflow”。

4 个工作流的名称“业务建模”、“需求”、“分析”、“设计”沿用了以往各个方法学的用语，这些用语也不严谨。

例如，“业务建模”末尾有一个“建模”，其他 3 个却没有，难道其他 3 个不是建模？如果把其他 3 个也加上尾巴“建模”，就变成批量刷废话了；如果把“业务建模”的“建模”去掉剩下“业务”，意思已经不对。

甚至“业务”二字都是模糊用语，这个后文再详述。

更贴切的名称可能是：

A-组织改进

B-系统责任

C-核心域逻辑

D-实现

但再三权衡，还是沿用之前的用语。

如果读者真正理解了 ABCD 的内容，叫什么问题都不大，改成 A-阿猫、B-阿狗、C-阿鸡、D-阿鸭也无所谓。

阅读到此处，读者可能会发现，本书中的“需求”和“设计”两个术语有两种用途。

一种用于表达建模得到的结果，例如“需求和设计不是一一对应的”；另一种用于表达建模的工作流，即“B-需求”工作流和“D-设计”工作流，例如“我正在做需求”。

希望下面的话能帮助理解：

为了得到需求，需要做的建模工作流有“A-业务建模”和“B-需求”，为了得到设计，需要做的建模工作流有“C-分析”和“D-设计”。

另外也说一下本书关于人员的一些用词，其中大多用词含义模糊。本书在某些场合沿用了这些词汇，但在此说明其含义。

(1) 开发人员

这个用语的含义目前已经模糊。有的人使用时含义很窄，意思等同于“编码人员”，有的人使用时含义很宽，泛指软件开发团队中的所有

人（当然，什么叫“软件开发团队”也是模糊的）。

本书作者在使用“开发人员”一词时，含义是：

软件开发团队中从事 ABCD 工作流中任何一项工作的人员。根据后文的讲述，所谓“测试”工作也是 ABCD 工作流，所以，测试人员也属于开发人员。

团队中从事管理工作的角色，不属于开发人员。

(2) 建模人员

“开发”本质上就是建模，“建模人员”一词所涵盖的人员和“开发人员”相同。

本书作者在描述某人使用 UML 或类似表示法的场景时，可能会使用“建模人员”一词来称呼该人员。

(3) 程序员

本书作者使用“程序员”时，含义是“实现人员”，即从事 D 工作流的人员。

(4) 产品经理、架构师

另外，本书可能还会出现“产品经理”、“架构师”等用词，但一般会出现在描述某个软件开发团队的某个场景时。产品经理一般指从事 AB 工作流的人员，架构师一般指从事 CD 工作流的人员。

本书作者在讲解知识或讨论问题时，会尽量使用严谨的用语，**不会主动使用**“产品经理”、“架构师”等含义模糊的用语。

1.2.3 逃不掉的思考

在迭代周期中，如果要追求好的结果，按 $A \rightarrow B \rightarrow C \rightarrow D$ 的顺序来进行推理是必须的，而且各个开发团队一直都在这样做。我们来看几个场景：

开发团队甲，所有成员都认真学习《软件方法》和认真做题，引进了 UML 和建模工具（例如 Enterprise Architect, EA），并按照书中的改进指南改进，最终得到很不错的结果。

开发团队甲的 $A \rightarrow B \rightarrow C \rightarrow D$ 很容易看出来。

开发团队乙，没听说过《软件方法》，也不用 UML 和 EA，用的是技术总监自己归纳的一套“软件工程方法学”和符号，也还算行之有效。

开发团队乙的 $A \rightarrow B \rightarrow C \rightarrow D$ 也容易看出来，但形式上和开发团队甲不同。

开发团队丙，快乐拥抱挂着“敏捷”或“领域驱动设计”名头的伪创新。这些伪创新投资少，见效快，门槛低，产量大，仪式感十足。开发团队轻松愉快、热热闹闹地讨论和糊墙。

这里面有 $A \rightarrow B \rightarrow C \rightarrow D$ 吗？也许有一点，但很少，这些热闹只是装模作样。真正起作用的 $A \rightarrow B \rightarrow C \rightarrow D$ 推理，可能是在编码的时候在大脑里朦朦胧胧进行的，推理的质量可想而知。

开发团队丁，觉得既然 ABC 我都不会，我也不想装模作样，就一步到 D，直接编码。

这种情况下, 还有 $A \rightarrow B \rightarrow C \rightarrow D$ 吗? 有的, 和开发团队丙差不多, 真正起作用的 $A \rightarrow B \rightarrow C \rightarrow D$ 推理, 是在编码的时候在大脑里朦朦胧胧进行的, 同样, 推理的质量可想而知。幸好, 省去了伪创新装模作样的成本。

★更巧妙的是, 有的开发人员以“敏捷”为名故意选择这样做, 目的是为了遮掩自己 ABC 能力的不足。他宁可直接跳到 D, 然后面对着编码环境慢慢在大脑里朦朦胧胧地补课前面的 $A \rightarrow B \rightarrow C$ workflow, 因为这样做“压力”会小很多。

★此时明面上的任务是编码, $A \rightarrow B \rightarrow C$ 的推导是顺带做的, 属于添头, 做到什么程度不好苛责。如果专门设置了一个任务, 摆明现在是做 A-业务建模, 那大家对产出结果的要求就高了, 那可就不好遮羞了。

开发团队戊的故事 (纯杜撰, 勿对号入座) 看起来有点不一样。他们所在的公司赶上了风口, 成了某个领域 (例如预制菜) 的互联网巨头。在开发和维护预制菜相关的项目时, 开发团队戊的领导觉得现在的前端框架 Vue、React 等还差点意思——更有可能是为了自己的简历积极主动地“觉得”Vue、React 等还差点意思, 于是搞了个内部项目, 自研前端框架“闪电五连鞭”。过了几年, 也许是风潮过了, 也许是历史使命已经完成, 预制菜市场冷却, 反倒是“闪电五连鞭”框架误打误撞, 受到全世界开发人员热捧, 成为公司的生存支柱。

这里面有 $A \rightarrow B \rightarrow C \rightarrow D$ 吗? 看起来像是颠倒了?

再看一个特别的, 程序员张三自己做一个东西给自己用。张三只会用编码工具, 也没学过 UML 或类似的知识。

张三这里面还有 $A \rightarrow B \rightarrow C \rightarrow D$ 的推理吗？

最后这两个，答案也是一样的，并没有什么特别，具体留到后文详述。

总之，每一个软件项目，只要不是故意摆烂，开发团队都会有 $A \rightarrow B \rightarrow C \rightarrow D$ 的推理过程——也许无意识、隐式地做，也许有意识、显式地做；也许推理过程严谨合理，也许推理过程漏洞百出；也许分成很多人来做，也许一个人做；推理产物的形式也许是 UML 模型，也许是其他.....

就像一道复杂的数学填空题，答案是 974/2025，如果考生能填对这个正确答案，他必然在考试过程中的某个时间点做了正确的推理——毕竟蒙对这样的填空题答案还是挺难的。只要不是故意摆烂，即使是学渣，也会尝试着去推理。只不过相对于学霸的一击必中，学渣可能要反反复复“敏捷探索”多次才答对，甚至直到考试结束还没答对。

本书希望教给读者一种严谨和高效的推理方法。当然，要掌握它，需要付出辛勤和汗水。

1.2.4 不了解 ABCD 的危害

1.2.4.1 思维颠倒

如果软件开发人员对以上的“A-业务建模”、“B-需求”、“C-分析”、“D-设计” workflow 没有概念，就会出现这样的现象：

问一名开发人员“你在做什么”，他可能回答“我在做设计”、“我在写文档”。

其实，此时他的大脑可能正在思考组织的流程（A-业务建模），或者在思考系统有什么功能性能（B-需求），或者在思考系统要封装的领域概念之间的关系（C-分析），但他通通回答成“在做设计”、“在写文档”。

也就是说，在他的大脑中，软件开发工作被简单地分为“写代码”和“做设计（写文档）”两部分。如果他没有在“写代码”，那么通通是在“做设计（写文档）”。

那么，他是怎么区分自己在“写代码”还是“做设计（写文档）”呢？可能是这样区分的：在 Visual Studio、Android Studio、Eclipse.....中写出来的叫“代码”，在 Word、wiki、Visio、EA.....中写或者画出来的叫“文档”。

这时候如果有人忽悠一句口号：代码就是设计，那就更妙了。因为本来我就认为“设计”是“代码以外的各种东西”，现在又说“代码就是设计”，一推导，不就变成了“代码就是（软件开发的）一切”？

即使没有“代码就是设计”这样的忽悠，开发人员也会把“设计”（注意，此处的“设计”不是本书严格定义的 D 工作流，而是“代码之外的所有东西”）或“文档”看成“代码”的一种比较概要或比较形象的表现形式。不同的“设计”或“文档”代表着“代码”的不同视图，可以让开发人员从不同的视角观察代码，如图 1-6 所示。



图 1-6 误解：文档只是代码的视图

这样的误解不只“普通”的开发人员会有。Martin Fowler 所著的 UML 畅销书《UML 精粹》，认为 UML 有三种用法：草稿、蓝图和编程语言，也是把 UML 模型看作是代码的视图——这是错误的。虽然 Martin Fowler 在某些社群的心目中如大神一般存在，但是从 Fowler 写作的其他书籍《重构》、《企业应用架构模式》、《分析模式》等可以知道，他的研究工作集中在“C-分析”和“D-设计”工作流，较少涉及“A-业务建模”和“B-需求”工作流。他关于“A-业务建模”和“B-需求”工作流的言论，应谨慎看待。

更危险的是，把“文档”当作“代码”的视图还会带来思维颠倒：先拍脑袋编码（快进到 D 工作流），然后再从代码（D）来反推 A、B、C 工作流的内容，导致其他工作流变成徒有形式的装模作样。

以下是几段我经历过的思维颠倒的对话：

★对话一

我：这个不应该是系统的用例（如果读者不理解什么叫“用例”，就先把它理解为“功能”好了）。

开发人员：是的！我都写好了，运行一下给你看，这个系统确实提供了这个用例。

系统是否应该有这个用例，应该从“A-业务建模”来推理，通过愿景、业务序列图等步骤的推理后，觉得应该有，系统就有，不该有就没有。不能说，我写好了代码，所以就应该有。

★对话二

我：这两个类的关系不应该是泛化，而是关联。

开发人员：是泛化，不信我打开代码给你看（或：逆向工程转出类图给你看）。

是否泛化关系应该从领域逻辑来判断，领域逻辑不是，那就不是，代码就不应该那样写。不能先写代码“人是猪的一种”（这肯定能通过编译器的检查），再用写好的代码来证明“人是猪的一种”。

★对话三

我：A 聚合（组合）B，这个不太对。

开发人员：对的，你看我代码，A 是聚合根，其他对象调用时要先找 A，存取数据也是以它作为单元。

同对话二，是否聚合（组合）关系应该从领域逻辑来判断，领域逻辑不是，那就不是，代码就不应该那样写。

如果了解了 ABCD 工作流的概念，以及在不同工作流中所需要思考和表达的内容，我们可能就不会再说“我在写文档”这种话了。“我在写文档”只是表达“我写的东西不是代码”或者“我正在用文档编

辑工具在工作”。

你在写什么“文档”？“A-业务建模”？“B-需求”？“C-分析”？我不写，我画图，难道不可以吗？我不写不画，我用语音清楚地表达组织的流程，难道不可以吗？我用 Word 编码（即 D-设计），难道不可以吗？我用 C#写需求（即 B-需求），难道不可以吗？

更有意义的说法应该是“我在做业务建模”、“我在做需求”如果说“文档”二字可以给你带来不可替代的快感，可以说“我在写业务建模文档”。

1.2.4.2 胡乱应对

开发人员常说要“应对变化”，甚至有的人还喊口号“拥抱变化”，但是很多人并不清楚要应对的是什么样的变化，也不知道应该怎样正确地应对变化。

当我们谈到“变化”的时候，真相有可能是下列之一：

真相（1）：根本没有变化

系统的需求（功能、性能和约束）没有变化，只是之前由于实现人员的能力问题，导致实现没有满足需求，或者能满足需求但自认为系统的内部结构不合理。这样的修改不能叫“应对变化”，只能叫“纠错”。

可笑的是，有的人也把这个情况也称为“应对变化”。就像考试有一道题难度稍高，学渣“敏捷”地算了一下，没算出来，把原来的擦掉，再来，还是没算出来，于是大喊“变化剧烈！”。

这种情况下，需要关注的工作流是“C-分析”和“D-设计”。

真相 (2)：系统的功能没变，性能和约束变了

例如，响应时间需要缩短，实现平台需要更换等。

应对这样的变化，需要关注的工作流是“D-设计”，实现的套路要调整。

真相 (3)：系统功能的合理变化

经过严谨的“A-业务建模”和“B-需求”的推理，认为系统的功能（用例、步骤、输入输出信息、业务规则.....）确实需要变化。

应对这样的变化，需要关注的工作流是“C-分析”。

注意，这里说的是经过“A-业务建模”和“B-需求”的严谨推理得出的“合理变化”。系统的功能之所以需要变化，根源在于系统所处的大环境中的某些东西发生了变化，而这样的变化往往是有其规律的。通过“C-分析”工作流，调整系统的核心域模型，使其更好地体现核心域的规律，有助于应对这样的变化。

真相 (4)：系统功能的不合理变化

这就是平时常说的“假需求”。和(1)一样，这其实不是“变化”，是“错误”。

没有经过严谨的“A-业务建模”和“B-需求”的推理，拍脑袋得到的所谓“系统功能”，不能改进组织流程，也不满足涉众利益。

这种情况下，关注“C-分析”工作流意义并不大，因为这种变化缺少规律。

如果我们发现在“C-分析” workflow 下功夫对“应变”的改进并不大，那意味着我们需要关注的工作流应该是“A-业务建模”和“B-需求”。先学会严谨推理出系统的功能，才能知道真相到底是（3）还是（4）。

我们用医生给患者看病类比：

真相（1）：

医生的诊断没问题，开的处方也没问题，护士或患者执行医嘱时执行错了。

这不是“变化”，这是“错误”。如果还把这个叫作“拥抱变化”，就是无耻了。

真相（2）：

输液时，换了另外一种品牌的一次性输液器。

真相（3）：

患者病情确实有变化。张三原来只是乙肝，现在是肝癌。

这种变化是有规律的。张三得了乙肝却不注意保养，依然酗酒和熬夜，很快进展到肝硬化，然后进展到肝癌。

如果充分了解肝脏的工作机制（C-分析），当张三被诊断出乙肝时，就可以合理应对当时的情况，并预测和提前应对后面可能会发生的变化。

真相（4）：

医生诊断错了病情。

张三出现恶心、乏力、食欲减退。村里的道士九叔给他诊断，认为他被鬼上身了，需要搞一个驱魔仪式。九叔已经精研驱魔理论体系和实践多年，一手辟邪剑法已经达到传奇王者。

实际上张三是得了乙肝，九叔的传奇王者对此没有帮助。

当然，如果是玩驱魔游戏或 Cosplay，又不一样了。

也许是不了解其中区别，也许是为了掩盖自己的无能，开发人员经常胡乱应对。

例如，把真相（4）说成真相（3），营造出“需求变化剧烈”的假象，从而掩盖自己缺少“A-业务建模”和“B-需求”能力的事实。

例如，对真相（3）和真相（4）避而不谈，只谈真相（2），或者想通过真相（2）的应对方案“D-设计”来应对真相（3）和真相（4），因为“D-设计”是他唯一熟悉的内容。

一些资料以“领域驱动设计”为名，结果一看内容，所举的例子就1-2个“领域”类，然后就开始讨论 Entity、Service、Repository、DTO、六边形架构.....哪里有“领域”？分明说的是“企业应用架构模式”、“互联网系统架构模式”。

在各种软件开发技术大会上，也可以看到这样的场景：某电子商务网站的架构师上台讲了一通，接着某视频网站的架构师上台也讲了一通，咦，这两人是同事吗，演讲内容如此相似？原来，他们讲的内容都是“D-设计”的内容。究其原因也许不是不愿意讲，而是讲不出来——该架

构师就会这一点点，连自己所开发系统的核心域都研究得很浅。

1.2.4.3 伪创新泛滥

很多开发人员只有 D 的知识。当岗位发生变化，需要他做 A、B、C 的工作时，按道理应该去认真学习 A、B、C 的技能才对。

可惜，很多人并不愿意走出自己的舒适区，甚至还会有意无意地把其他人拉到自己的舒适区。

例如，在和涉众讨论需求时，频频蹦出一些“技术潮词”，目的就是以自己的“所长”来碾压涉众，从而掩盖自己业务建模（A）和需求（B）技能的不足。

例如，在讨论核心域的类模型（C）时，动不动就谈到如何实现（D）或者质疑“会不会有性能问题”（D），从而掩盖自己抽象能力的不足。

于是，各种投其所好的伪创新就登场了。

有的伪创新极力贬低 A、B、C 的重要性，通过“砸烂一切枷锁”来吸引热血青年。例如，想那么多有啥用，最后不是还得写代码？张嘴就是 Linus Torvalds 的“Talk is cheap. Show me the code.”。

后来，“发明家”及其追随者慢慢发现砸烂一切是不行的，追随者的信念开始动摇。于是，伪创新不再贬低 A、B、C，而是从 D 来臆想 A、B、C，得到的 A、B、C “方法学”非常“简单易学”，让只了解 D 的开发人员感觉“很受用”。

例如，深入第一线调研各类涉众的利益很麻烦。有办法，摆一个“现场客户”在旁边，开发人员就可以心安理得坐在电脑前面编码，有问题

就推给“现场客户”。

例如，认真学习领域知识的各种概念和术语很麻烦。有办法，开发人员可以按照自己的理解创造一套“通用语言”。

伪创新往往并不会直接说自己简单易行，而是会说自己很高深。宣传中往往带有“艺术”、“禅”、“道”等字眼，有意无意地朝宗教、艺术、玄学方向引导——比起枯燥的数学理论和逻辑推理，这些东西可是太好下嘴了。还有一个很大的优势，**一些媒体人听到“艺术”、“禅”、“道”等字眼就亢奋**，自觉地加入到宣传伪创新的队伍中。

开发人员一开始以为很难很深奥，上手一学，发现其实不难！可以说是：投资少，见效快，产量高，门槛低，而且仪式感十足。最妙的是，不用走出舒适区辛苦学习，就得到了“方法学”，这可太符合只了解 D 的开发人员的胃口了！开发人员立刻**有捡到了便宜的感觉**，心中豪气顿生——不愧是我！别整三岁的，有能耐你整四岁的！

伪创新还会声明“领域驱动设计不是银弹”之类，也是为了进一步塑造形象。我都诚实地说了我不是满分，所以我前面塑造的 90 分的形象应该是真的。

关于各个工作流的各种伪创新，本书后面各章节还会进一步讨论。

1.2.5 没有测试工作流？

“测试”可以看作建模的验证过程，所以不能光说“做测试”，要清楚认识“测试”所验证的内容。

如果“测试”验证的是组织流程中各个系统之间的协作，那就是“A-

业务建模”。

如果“测试”验证的是目标系统的整体表现，那就是“B-需求”。

如果“测试”验证的是目标系统内部各个部件之间的协作，那就是“C-分析”和“D-设计”。

无论是启发、定义还是验证，如果你思考的内容是某一个工作流的内容，你就是在做该工作流。在一个迭代周期中，启发、定义和验证往往是交错进行的。

1.2.6 没有项目管理 workflow?

本书关注的范围限于方法学，即 $A \rightarrow B \rightarrow C \rightarrow D$ 的正确推导方法，至于推导是一个人做的，很多人做的，甚至是猫做的、狗做的、外星人做的，没有直接关系。

还是用前面的大楼类比。两幢大楼耸立在那里，地震来了，一幢塌了，另一幢没塌。

直接因素是大楼的结构、所用的材料、所在位置的地质环境等，这些涉及到艰深的工程力学、流体力学、岩土力学等知识——类似于本书所研究和阐述的方法。即使大楼是外星人盖的，也要讲这个基本法。

直接原因往往比较难理解，此处要提防，有的人为了掩盖自己的无能，干脆找比较容易理解的其他原因来把水搅浑。例如，一出事故就嚷嚷“有人吃回扣了”，就算经过调查没人吃回扣，他也会从工作服的颜色，工人是否结对洗澡，施工队开会时是否站立等更容易理解的方面来找原因。

当然，要成功完成一个软件开发项目，还有很多其他工作，例如进度管理、人力资源管理等，但这些都不属于本书讨论的范围。关于软件项目管理方面的内容，读者可自行阅读专门的书籍。

1.2.7 人工智能对 ABCD 各工作流的影响

从广义上来说，所有封装了逻辑的信息系统都可以叫“人工智能 (AI)”。例如，计算器封装了运算的逻辑，你只要告诉它输入： $2355465722232 \times 5465768797343$ ，它就会给你输出，而且比绝大多数人要快。所以，我们使用 AI，AI 在某些方面比人强，已经有几十年的时间了。

★算盘不是 AI，知识点见第 4 章。

当然，我们这里谈到的 AI 是狭义的，指生成式人工智能 (Generative AI)。自从 GPT-1 在 2018 年推出，生成式人工智能热潮兴起至今。

现在 (2026 年 1 月) 大多数人都已经在使用 Cursor 等 AI 编码工具，给 AI 提要求，AI 生成代码。甚至有造词 “Vibe Coding”，不要程序员理解 AI 生成的代码。

这相当于在 D (实现) 工作流应用 AI。

但是，由于没有把控住 C-分析的环节，随着系统需求的增加，AI 的各种修补、拼凑变得越来越笨拙，最终不可收拾。

于是又有人说：用 AI 要 Spec-Driven，相当于在形式上提了要求，但很多人（包括伪创新圈子以及伪创新 Fans 心目中的国外“大神”）

不了解 A-业务建模、B-需求、C-分析、D-设计在内容上的区别以及其中的推导过程，谈论 Spec 的时候，根本不知道自己在谈什么，这和之前这些人谈论 UML 时一模一样。

内容以前是怎么来的，现在还是怎么来的。以前是拍脑袋，现在还是拍脑袋。以前是想到什么写什么，边界跳来跳去，现在还是想到什么写什么，边界跳来跳去。

缺少方法学的知识，Spec 越来越像伪代码，其实就是重复之前非常受伪创新买家和卖家欢迎的“敏捷实践”，拍脑袋编码，然后从代码出发，需要的时候“补文档”。这就是前文所严厉批评的思维颠倒。

在 ABC 工作流，AI 目前带来的帮助并没有那么大——人类还没理清更大粒度的 ABC 工作流怎么做，刚取得一些进展，又有被伪创新拉着倒退的趋势。

现在又有一个不好的情况，伪创新圈子的“造词”、“念经”、“批量刷废话”已经在污染 AI。在 ABC 工作流，我们暂时无法把 AI 当成“专家”，只能把 AI 当成一个非常非常聪明的学生，用严谨的方法学来给它解毒并引领它。

本书的后续章节在探讨软件开发工作流的每一个步骤时，都会引入 AI 辅助的内容。我们会把方法学的知识细致分解并训练 AI，让它给我们带来真正的价值。

1.2.8 自测题



本书不提供习题答案，请扫码或访问 umlchina.com/book/quiz1_2.html 自测，做到全对自然就知道答案了。



1[单选]

以下描述最可能对应于软件开发中的哪个 workflow?

每个项目由若干活动组成，每项活动又由许多任务组成。一项任务消耗若干资源，并产生若干工件。工件有代码、模型、文档等。

☐ A) 业务建模

☐ B) 需求

☐ C) 分析

☐ D) 项目管理

2[单选]

以下描述最可能对应于软件开发中的哪个 workflow?

```
/// <summary>
/// Add money to this account
/// </summary>
/// <param name="amount">Amount of money to add</param>
public void CreditMoney(decimal amount)
{
    //Amount to Credit must be greater than 0. --> Domain logic.
    if (amount <= 0)
        throw new ArgumentException(Messages.exception_InvalidArgument, "amount");

    //Account must not be locked.
    if (this.Locked)
        throw new InvalidOperationException(Resources.Messages.exception_AccountIsLocked);

    //Credit means adding money to this account. --> Domain Logic
    this.Balance += amount;
}
```

☐A) 需求分析

☐B) 代码设计

☐C) 详细设计

☐D) 设计

3[单选]

以下描述最可能对应于软件开发中的哪个 workflow?

系统向会员反馈已购买商品的信息。

☐A) 功能规约

☐B) 需求分析

☐C) 需求

☐D) 需求规约

4[单选]

以下描述最可能对应于软件开发中的哪个 workflow?

在去给外婆拜年的路上,李雪琴突然想起需要一些现金给外婆做现

金红包。她先用高德地图查询了附近的 ATM 机网点，并挑中了一处工行的网点，开车过去一看，好家伙，路边这么多人排队取现金！李雪琴只好又看高德地图，挑了一家她觉得人应该会比较少的，开车过去，终于在 ATM 机取到了现金，然后，她又想了想哪里有卖红包的地方……

☐A) 业务建模

☐B) 用户故事

☐C) 需求

☐D) 流程分析

5[单选]

以下不属于建模工作流 ABCD 的是_____。

☐A) 需求

☐B) 分析

☐C) 设计

☐D) 测试

6[单选]

本书中，建模工作流的名称沿用了之前方法学中的名称，这些名称其实都不太合适，例如，_____工作流的名称改为“系统责任”更合适。

☐A) 业务建模

☐B) 需求

☐C) 业务需求

☐D) 分析

7[单选]

有方法学家发明了比“糊墙”还有仪式感的“美女建模法”。

例如，某医卫领域软件公司招聘一批颜值较高的美女，分别扮演患者、护士、医生、收费员、HIS 系统、支付宝、微信等。和客户沟通时，就让这些美女一起上场表演各种场景给客户看。

客户一致反映：这样的方式生动、活泼、新颖、看得见、闻得着、有嚼头、有回味、一举多得（附加题：这一串的出处是？），下次沟通还这样搞！

请问。美女们向客户表演的最有可能是哪个工作流的内容？

☐A) 业务建模

☐B) 总体设计

☐C) 需求

☐D) 分析

8[单选]

软件开发工作流：A-业务建模、B-需求、C-分析、D-设计。目前（2026 年 6 月），AI（人工智能）帮助最大的是哪一个？

☐A) A-业务建模

☐B) B-需求

☐C) C-分析

☐D) D-设计

1.3 统一建模语言 UML

1.3.1 UML 的历史和现状

UML (Unified Modeling Language, 统一建模语言) 是由 OMG (对象管理组织) 维护的一个软件建模表示法标准。

在 1.2 中, 我们阐述了 $A \rightarrow B \rightarrow C \rightarrow D$ 的推导是不可避免的, 但具体如何推导, 有各种不同的做法, 这些做法可以称为“方法”。甚至只要愿意, 每个人都可以创造自己的“方法”, 无非是有的正确, 有的错误, 有的高效, 有的低效。有一些“方法”被成体系地归纳出来, 并向业界推广, 这些可以称为“方法学”。

最开始的软件开发方法学重点关注的是 D 部分, 即所谓的“程序设计方法学”。后来, 才逐渐在方法学中加入前面的部分, 大致的添加顺序和推导的顺序刚好相反, $\rightarrow C \rightarrow B \rightarrow A$ 。其中的很多概念借用了其他学科的术语。像“流程建模”、“需求”等术语在计算机出现之前就已经存在, 而且含义和今天软件开发中使用时的含义差不多。

本书不想花很多篇幅来回顾这些方法学中的概念的历史, 感兴趣的读者可自行搜索相关论文, 例如 Kolligs 等人写的 “The Origins of Requirements” [Kolligs 2021]。

20 世纪 60-80 年代, 有名的软件方法方法学有: 功能分解、数据

流、E-R（实体-关系）等。

进入 20 世纪 90 年代，OOAD（面向对象分析设计）方法学开始受到青睐，许多方法学家纷纷提出了自己的 OOAD 方法学。流行度比较高的方法学有 Booch、Shlaer/Mellor、Wirfs-Brock 责任驱动设计、Coad/Yourdon、Rumbaugh OMT 和 Jacobson OOSE。其中，Jacobson 的方法学添加了用例、业务工人、业务实体等概念，为 OOAD 方法学扩展了业务建模和需求部分。

这个百花齐放的局面带来了一个问题：各个方法学有自己的一套概念、定义和标记符号。

例如现在 UML 中的操作 (Operation)，在不同方法学各有叫法，这些叫法有：责任 (Responsibility)、服务 (Service)、方法 (Method)、成员函数 (Member Function)

同一个类图，不同方法学也有各自的符号表示，如图 1-8 所示。在图中，我们可以看到，三角形符号在 OMT 方法学中表示泛化，在 Coad/Yourdon 方法学中却表示关联，Coad/Yourdon 方法学中泛化用的是类似铃铛的形状。

类似这样的差异造成了混乱，使开发人员无从选择，也妨碍了方法学的推广。

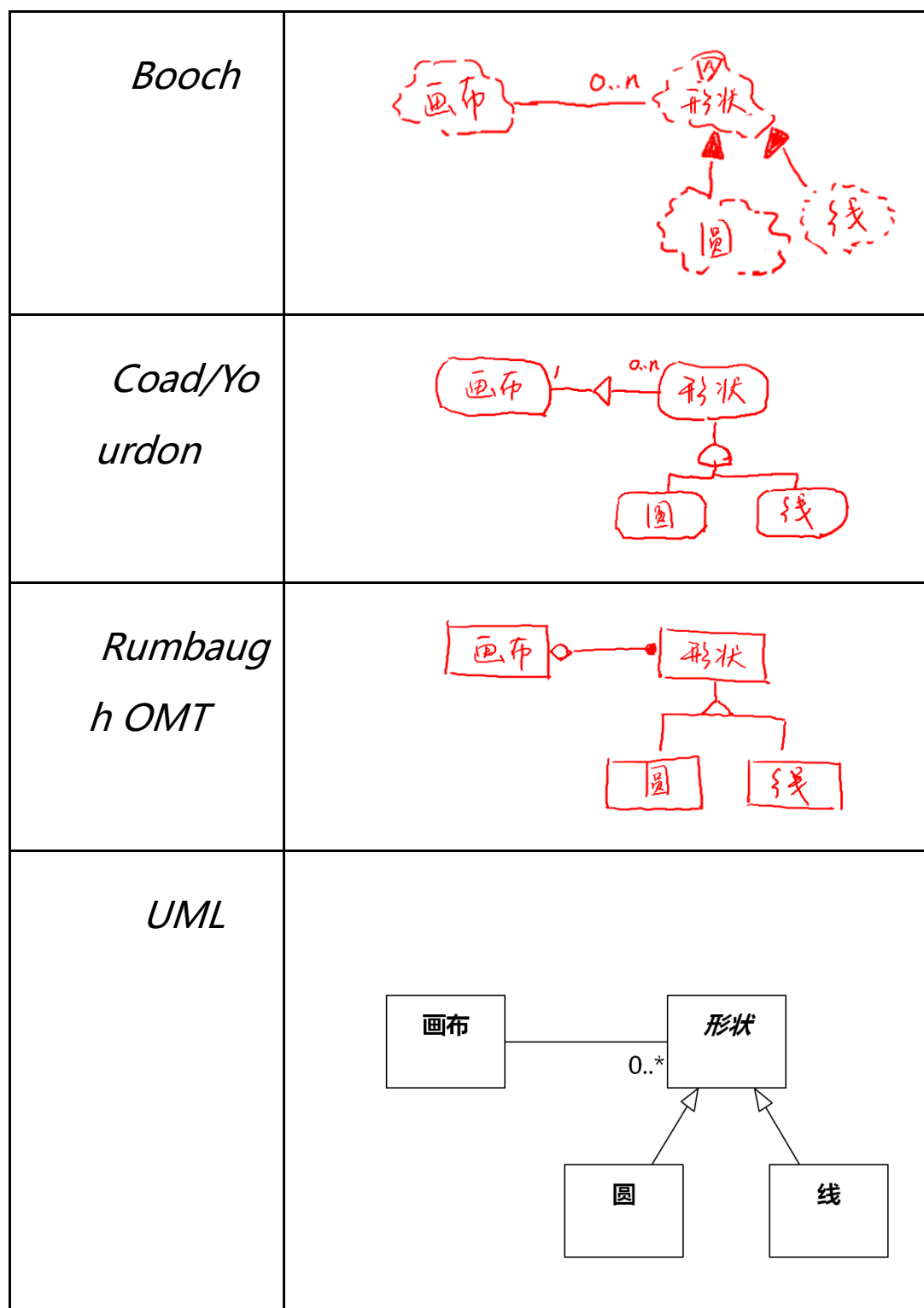


图 1-8 不同方法学图形比较

1994 年，两位已经在 Rational 公司工作的方法学家 James Rumbaugh 和 Grady Booch 开始合并 OMT 和 Booch 方法。随后，

Ivar Jacobson 带着他的 OOSE 方法学加入了 Rational 公司，一同参与合并工作。这项工作造成了很大的冲击，因为在此之前，各种方法学的拥护者觉得没有必要放弃自己已经采用的表示法来接受统一的表示法。

Rational 公司的这三位方法学家被大家称为“三友”（three amigos）。1996 年，三友开始与 James Odell、Peter Coad、David Harel 等来自其他公司的方法学家合作，吸纳他们的成果精华。1997 年 9 月，所有建议被合并成一套建议书提交给 OMG。1997 年 11 月，OMG 全体成员一致通过 UML，并接纳为标准。

从 2005 年起，UML 被 ISO 接纳为标准。ISO/IEC 19501 相当于 UML 1.4.2，ISO/IEC 19505 相当于 UML 2.1.2。2012 年，ISO 继续接纳 UML 2.4.1 为 ISO/IEC 19505-1:2012 和 ISO/IEC 19505-2:2012，接纳 OCL 2.3.1 为 ISO/IEC 19507:2012。

2011 年，中华人民共和国也发布了统一建模语言国家标准 GB/T28174。

UML 的最新版本是 OMG 于 2017 年 12 月通过的 UML 2.5.1，相关网址：<https://www.omg.org/spec/UML/>。

OMG 还和各种行业标准组织如 DMTF、HL7 等结盟，用 UML 表达行业标准。

UML 诞生已经超过 25 年，在软件开发表示法标准上已经获得了胜利。随便打开一本现在出版的软件开发书，里面如果提到建模，使用的标准符号基本都是 UML。

另外，以 UML 为契机，掀起了一股普及软件工程的热潮，在 UML 出现后的几年，不但有关建模的新书数量暴增，包括 CMM/CMMI、敏捷过程等软件过程改进书籍数量也出现了大幅度增长。制定 UML 标准的角色（OMG）、根据标准制作建模工具的角色（UML 工具厂商）、使用 UML 工具开发软件的角色（开发人员）这三种角色的剥离，也导致建模工具的数量和种类出现了爆炸性的增长。而之前的数据流等方法从来没有像面向对象分析设计方法一样，出现 UML 这样的统一表示法，从而带动大量书籍和工具的产生。

最开始一批 UML 书籍，基本上由方法学家所写。最近几年，以“UML”为题的新书大多为高校教材或普及性教材。这并不是说 UML 已经不重要，而是没有必要再去强调，焦点不再是“要不要 UML”，而是要不要建模、如何建模。

1.3.2 使用 UML（或某个标准建模语言）的理由

在开发团队中，不乏刻意排斥 UML 的人。这些人如果只是不使用 UML，改为使用其他标准的图形表示法（如 BPMN），那倒没有什么好说的——但仔细观察可以发现，绝大多数情况下并非如此，这些人要么使用自编的“图形表示法”，要么使用文本，甚至有的人只强调“口头交流”。

当然，通过 UML、自编图形、文本、语音等形式都可以交流软件开发中的各种逻辑，但能达到的目的和付出的代价是不一样的。

1.3.2.1 听觉 vs. 视觉

回忆一下我们在学生时代做听力题和阅读理解题的过程,就可以体会到,相对于听觉,视觉传递信息的效率更高,而且可以传递更复杂的信息。正常情况下,把模型表达成视觉信息,不管是文本还是图形,比起语音信息来说是更好的选择。非正常情况,视觉无法使用的时候,例如开车或者累得眼睛睁不开,这时用语音来见缝插针,通过听觉利(zhà)用(gān)建模人员的生产力,也不是不可以。

如果有人以“敏捷”为理由,特别推崇“口头交流”,排斥文档或图形,很可能是遮羞布,背后的**脓包**是此人没有能力剖析复杂逻辑,试图通过这种方式来遮掩。

有的开发团队动不动就开会,你一嘴我一嘴,场面看起来热热闹闹,其实沟通的效果不好,更谈不上思考的深度和知识的沉淀。对比一下街坊老大爷下象棋的热闹和职业棋手比赛时的沉静就知道了。

1.3.2.2 文本 vs. 图形

相对于只有自上而下顺序的文本(包括自然语言文本和编程语言文本),能够朝四个方向扩展的平面图形(如果有三维模型就更好了)更容易让建模人员看出概念之间的联系。例如,图 1-9 和图 1-10 的内容,如果没有图形的帮助,通过文本一行一行地构造和阅读模型,人脑的负担非常重。

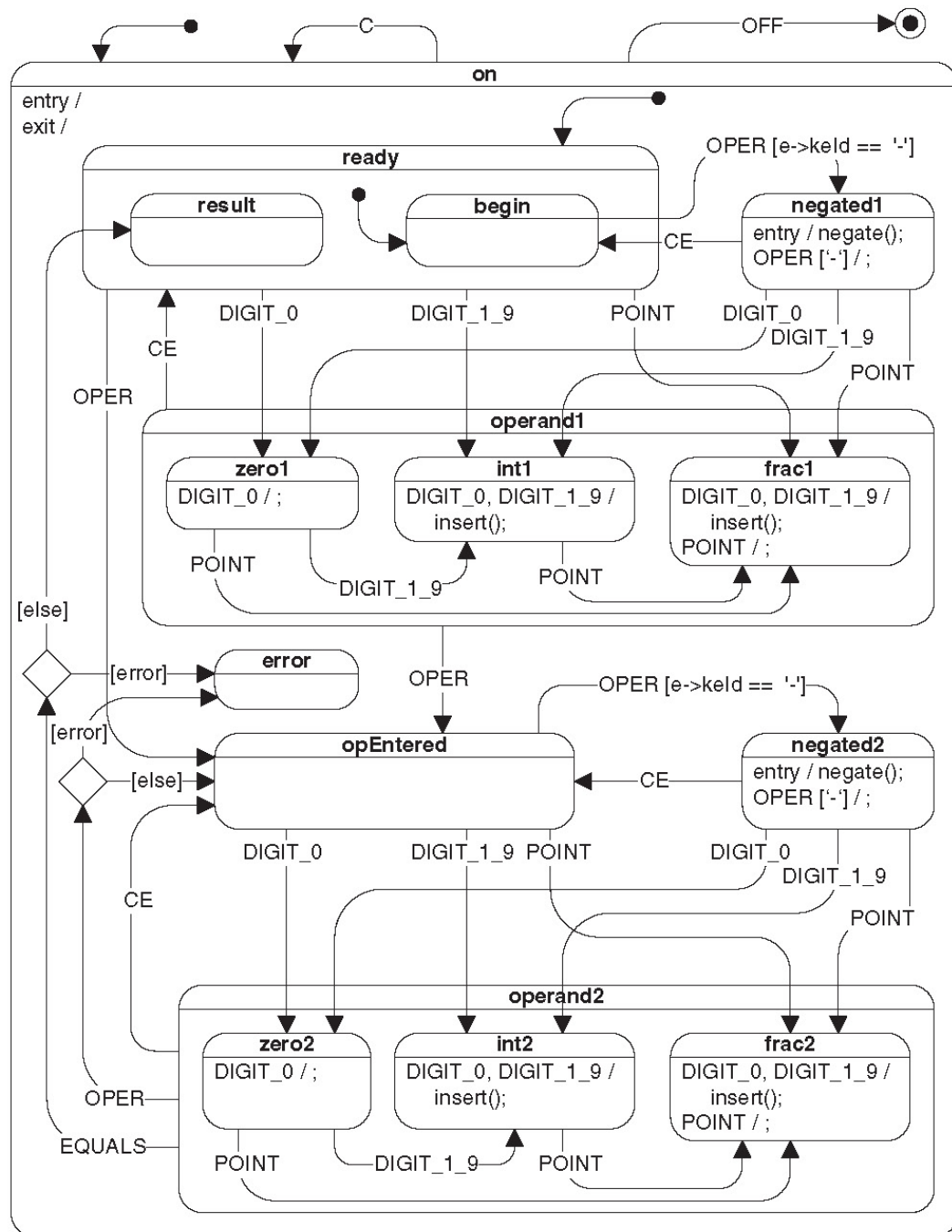


图 1-10 计算器的状态机图，摘自 Practical UML Statecharts in C/C++[Samek 2008]

说到这里，又不可避免地要提醒，故意选择文本的形式来表达领域知识，有可能也是一种遮羞布。图 1-9 和 1-10 的内容如果用文本表达，可能会得到很多页文本——这就有了理由：因为工作量太大了，所以很多地方无法做深入的思考，可以原谅！

1.3.2.3 自编图形 vs. 标准图形

事实上，纯口头交流甚至纯文本交流是很少见的。除非要交流的问题极其简单或者其他同事对这样的人极其容忍，否则他至少也得随意画几张“草图”来传达自己的意思——自编的图形才是比例最大的存在。

这样的人之所以用自编的图形，往往并不是因为他看出了 UML 有哪些不足，然后用自己的表示法弥补了这些不足，而是要遮掩背后的一些脓包。

脓包一：利益绑架

项目中，有人画了张自编的图形，然后往往会伴随这样的招呼，“来，我给大家讲讲！”。这意味着项目要依赖于“我”头脑中的隐式知识——要是“我”不给大家“讲讲”，大家就不理解“我”画的图的意思，项目就玩不转了。于是，“我”在团队里的地位就提高了，大家得哄着“我”捧着“我”，领导不敢轻易开掉“我”，“我”提出离职领导还要挽留。这在有一定资历、但又不对项目的成败承担首要责任的“高手”身上表现得更明显。

★开发人员让我看他的模型时，如果开口说“我先来给你讲讲”，我都会拦住，“如果还需要你先讲讲，说明你所想的没有体现在模型中”。

脓包二：遮羞

因为符号是“我”自创的，所以这个图的解释权归“我”所有。如果有其他同事质疑图上什么地方画得不对，“我”就有了充分的躲闪空间——“你理解错了，这个图形不是你以为的那个意思。你以为我画

的是鹿，其实这是马，在我的规范里面，马就是鹿的意思，鹿就是马的意思”。

如果使用了标准的表示法，例如用 UML 画了一个状态机图，其他人就有得说了，“好像手册上不是这样说的”，“我看那个书上不是这样说的”，这不就尴尬了嘛。

自编图形未必是看起来十分简陋的白板草图，也有看起来比较精致的。例如，来自某项目的图 1-11。

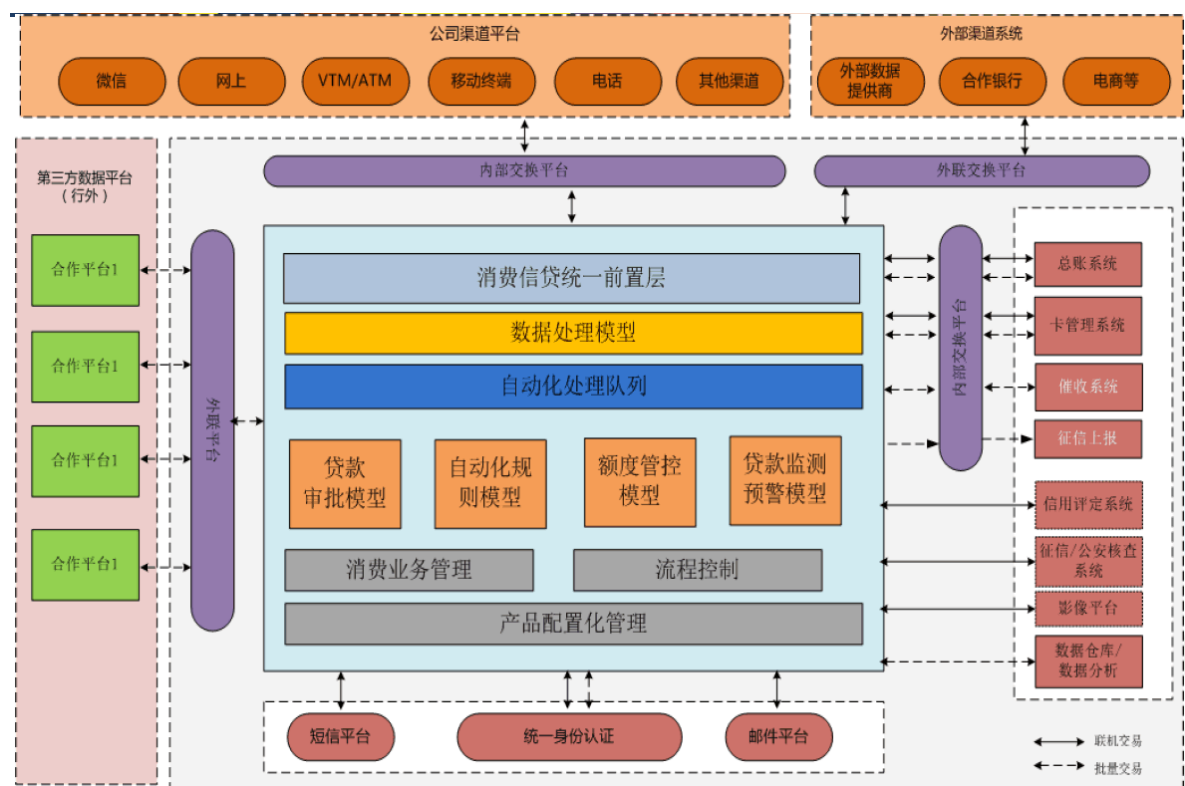


图 1-11 自编图形示例

其实，抛弃标准符号和建模工具带来的便利，用绘图工具或幻灯片工具画出这样的图，所花费的时间往往还要更多。但是，有意思的是，有的人画这样的图，还以“敏捷”为理由。

注意，上面说的只是“看起来比较精致”，其实内容还是很粗陋的，

仅仅是把一些动词、名词的罗列成一个个形状，各个形状之间的关系基本没有，读者可以把图 1-11 和图 1-9、图 1-10 对比。

有心的读者可以观察近年来被吹捧为“革命性创造”的领域驱动设计的文章，看看其中所画的图，看看会不会发现上面所批评的情况，简单罗列，无一致性。然而，为了掩盖逻辑上的不足，这些图的作者会把力气下在“美工”方面，因此，伪创新的图有时看起来比科研风格的图更赏心悦目，对外行来说更有吸引力。

1.3.2.4 挑破乱七八糟图的脓包

我们甚至可以“抛开事实（具体领域知识）不谈”，仅从“一致性”这一点入手，就可以挑破这种自编“乱七八糟图”的脓包。

自编“乱七八糟图”的画图者，很可能并没有归纳过各种形状的定义，只是凭感觉随意使用，或者即使归纳了，也不会像标准语言这么严谨，所以，大概率会产生“不一致”的问题。

我们就以图 1-11 为例。

图 1-11 中，同样被称为“平台”的东西，有着不同的形状和颜色，如图 1-12。

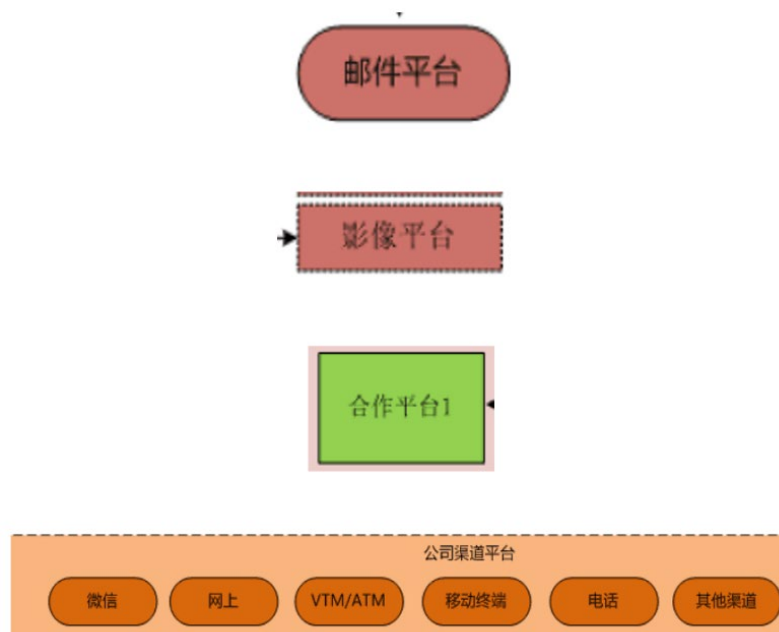


图 1-12 图 1-11 的不一致示例 1

我们再来看中间这些方框，如图 1-13。

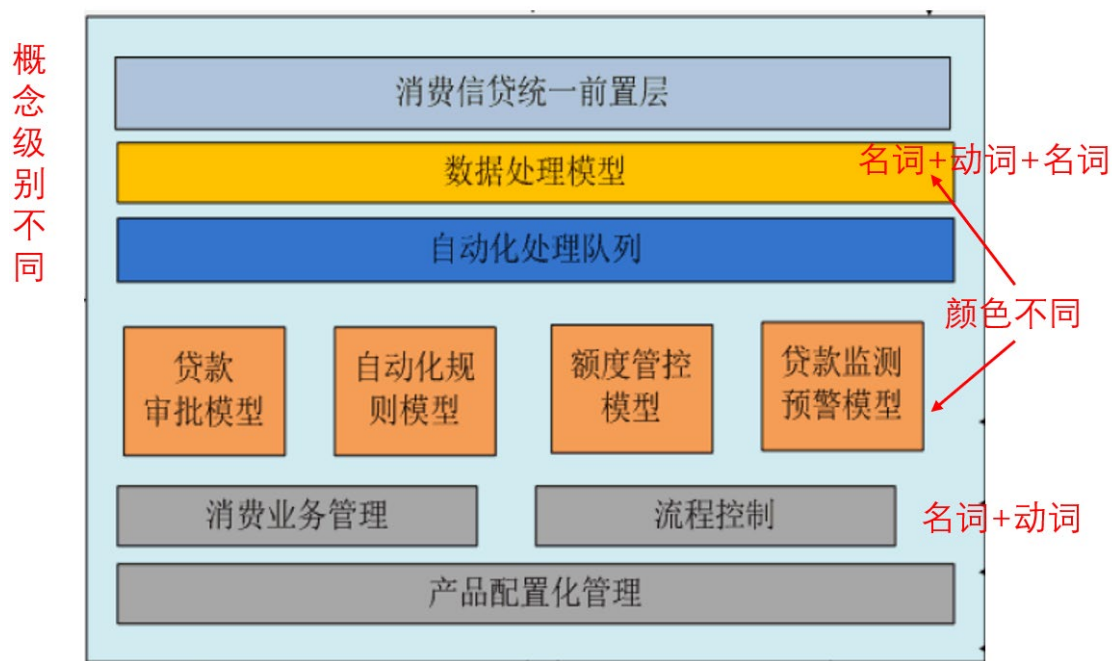


图 1-13 图 1-11 的不一致示例 2

先看图 1-13 的最顶上 3 个方框，它们形状一样，长度一样，但有的叫“层”，有的叫“模型”，有的叫“队列”，这些概念可不是一个级的。

别的。

（可能的辩解）也许在颜色中暗示？

再来看方框里的命名。最上面 4 行的命名，基本上都是“名词+动词+名词”结构，例如“消费信贷+统一前置+层”、“额度+管控+模型”，最下面 2 行的方框，命名是“名词+动词”结构，例如“消费业务+管理”、“流程+控制”。

（可能的辩解）你没脑子吗，自己脑补一个尾巴不行吗，“.....模型”、“.....层”、“.....功能”、“.....队列”、“.....AI 赋能领域驱动设计革命性创新微服务功能模块组件分布式大数据数智化平台”。

还有，名称同样是“模型”，颜色也有多种.....

读者可以尝试从“一致性”着手，看看身边的同事画的“乱七八糟图”有没有这方面的问题。

可能有人会问：难道用 UML 来画就没有这个问题吗？

有的，例如，建模人员故意往用例的圈圈里乱填东西，有的填名词，有的填动词，有的填形容词，但这是违反 UML 相关的规范或指南的，其他人可以看出问题，批评和纠正。刚才说的情况是没有规范或规范模糊，这两者是不一样的。

用法律类比：精确严谨的法律条文并不能保证没有人违法，但至少大家有共识，什么是违法，什么不是。而另一种情况则可能是，孪生兄弟张三和张四做了同样的事情，张三违法，张四不违法，理由？不知道。

1.3.2.5 基本共识上的沟通

表示法标准并不是随便哪个人拍脑袋定下来,然后毫无道理地强迫大家接受。符号背后往往隐含着我们对某个学科的一些基本共识。

如图 1-14,最上一行的积分表示法“ $\int$ ”,幼儿园小朋友也能画出这样的形状,但其所代表的知识可能需要上到大学才能理解。如果要懂得为什么是这样一个表示法,还需要了解数学史中莱布尼茨、傅立叶等人的贡献。图 1-14 中间一行的五线谱小豆芽和横线,最下一行 UML 的小人圈圈和框框,也是如此。

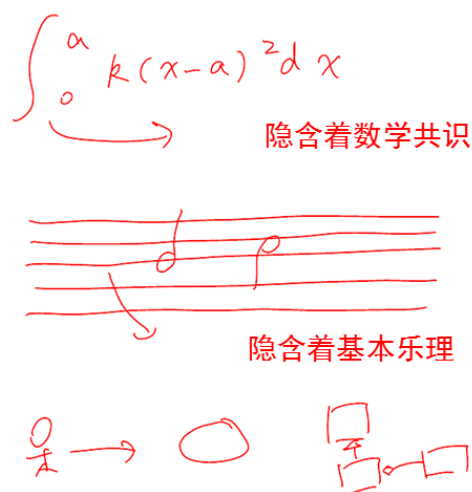


图 1-14 表示法背后隐含基本共识

这意味着,学习建模表示法标准并非硬生生把形状记住后照葫芦画瓢就行,还要学习背后的一些建模知识。但是,开发团队成员咬咬牙,花费一些精力跨过这个门槛是值得的,因为一旦有了基本共识,会大大提高沟通的效率和深度。在严谨建模思维的追问之下,有意无意遮掩的脓包也会被强制露出——这也是一些“高手”潜意识里不愿意直面 UML 的深层原因。

★面对一个棋局,下一步怎么走? 在业余棋手看来到处都是正确答

案，在职业棋手眼里，值得讨论的选项只有两三种，因为职业棋手针对一些东西形成了共识，大大减少了思考中的浪费。

★有的开发人员的“十年工作经验”实际上是“一年工作经验用了十年”，一直在热热闹闹的民工层次徘徊，没有积累和成长。

不过要注意一点：使用 UML 沟通仅限于软件组织内部，UML 模型不是用来和涉众沟通的！这个道理以及和涉众沟通的技能将在第 7 章详细叙述。

1.3.3 SysML

UML 的另一个成就，就是把“统一建模语言”扩展到了系统工程。INCOSE（国际系统工程协会）和 OMG（对象管理组织）以 UML 为基础，进行系统工程方面的扩展后推出了 SysML (Systems Modeling Language, 系统建模语言)，目的是和 UML 相似：为了统一系统工程的建模语言。

UML 是信息系统的建模语言。“信息系统”的意思是该系统只负责处理信息：信息流进来，经过系统的处理，变成信息流出去，但是，信息系统不能处理物质流和能量流——或者简单说就是能量流，因为物质是能量表现形式之一。

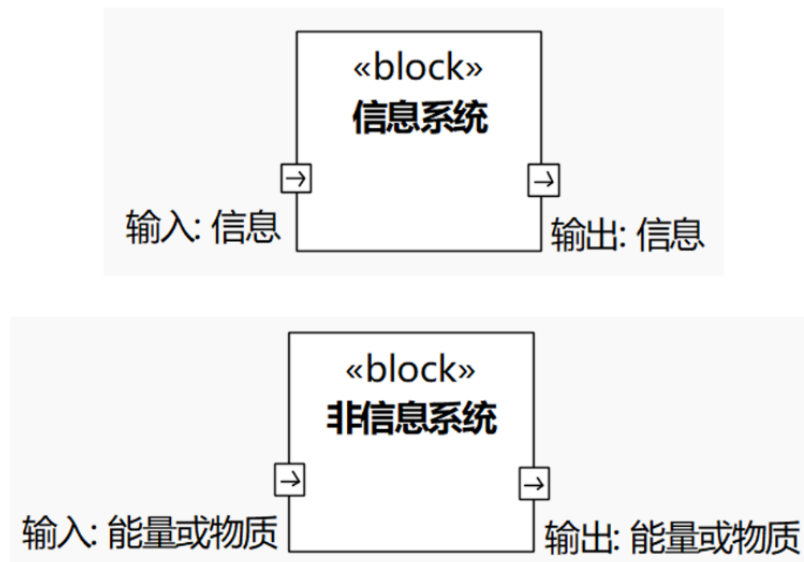


图 1-15 信息系统和非信息系统（此图用 SysML 表达）

SysML 可以建模处理能量（物质）流的非信息系统。将来能量（物质）、信息进一步大一统，SysML 是可以取代 UML 的，但目前还不能。

关于系统工程、软件工程、SysML、UML 的区别，我用刘慈欣的小说《流浪地球》举个例子。

太阳即将在 400 年内发生氦闪变成红巨星，人类社会怎么办？

众多科学家经过大量研究，得到一个解决方案：给地球装上发动机，驱动地球飞向比邻星（天寿啦！正好三体舰队迎面飞来，嘭！）。

小说《流浪地球》原文：

地球发动机安装在亚洲和美洲大陆上，因为只有这两个大陆完整坚实的版块结构才能承受发动机对地球巨大的推力。地球发动机共有一万二千台，分布在亚洲和美洲大陆的各个平原上。

.....

在六千米处，我们见到了进料口，一车车的大石块倒进那闪着幽幽红光的大洞中，一点声音都没传出来。

.....

“重元素聚变是一门很深的学问，现在给你们还讲不明白。你们只需要知道，地球发动机是人类建造的力量最大的机器，比如我们所在在华北 794 号，全功率运行时能向大地产生一百五十亿吨的推力。”

“地球发动机”是一个巨大的系统，横跨多个学科，能源、材料、建筑、物流、人员管理.....。

其中用来描述和处理信息的信息系统可能只占其中的一小部分，UML 不适合描述“地球发动机”，SysML 可以。

通过 SysML 建模，不断把“地球发动机”分解成各个 Block，Block 又分为小 Block，可能会得到其中一个 Block 叫“发动机中控系统”，这是一个信息系统。该系统其中一个功能可能是：根据接收到的测量数值(可能有上万个)，计算地球发动机下一步的最佳行动。我用 SysML 的块定义图画了图 1-16，只是简单展示，我也不知道地球发动机应该分成哪些 Block。

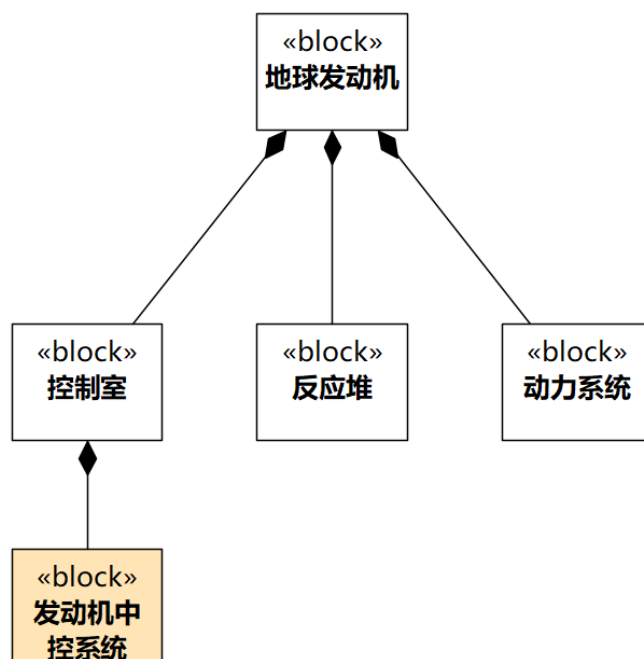


图 1-16 地球发动机的分解（此图用 SysML 表达）

图 1-16 中的“发动机中控系统”这个信息系统，就适合用 UML 来建模。

听起来 SysML 描述的系统比 UML 描述的信息系统大，是不是意味着 SysML 建模的难度更高？

目前来说并非如此。目前使用 SysML 建模时，更多的是“描述”，例如描述图 1-16 中“地球发动机”的各个部件及部件之间的关系。至于为什么地球发动机的部件和关系是这样的，这可能是其他各个学科的专家的工作成果。SysML 建模人员只需要通过模型如实描述并验证。

我们再来看图 1-16 中的“发动机中控系统”。即使 SysML 建模人员已经通过 SysML 建模严谨地推导出信息系统的责任，包括输入什么信息，输出什么信息，从输入到输出计算的规则是怎样的，也就是说把 B-需求都给描述清楚了，“发动机中控系统”这个信息系统的内部部件

以及部件之间的关系依然是不确定的，需要“发动机中控系统”的开发人员自己构思和实现——100 个团队可能有 100 个不同的构思。

“地球发动机”的部件以及部件之间的关系，可是其他专家团给出来的，并不需要自己构思。这一点就使得“发动机中控系统”的建模要比“地球发动机”更难。

当然，如果领导只是吩咐一句，做一个“地球发动机”，剩下所有工作都由 SysML 建模人员来做，那难度可就大了。

我们来看图 1-16 中的“反应堆”，《流浪地球》原文提到的“重元素聚变”应该属于“反应堆”的责任。我用 SysML 画了一张活动图，如图 1-17。可以看到，这个过程处理的是能量（物质），原料和催化剂进来，变成能量出去（当然，还有废渣）。

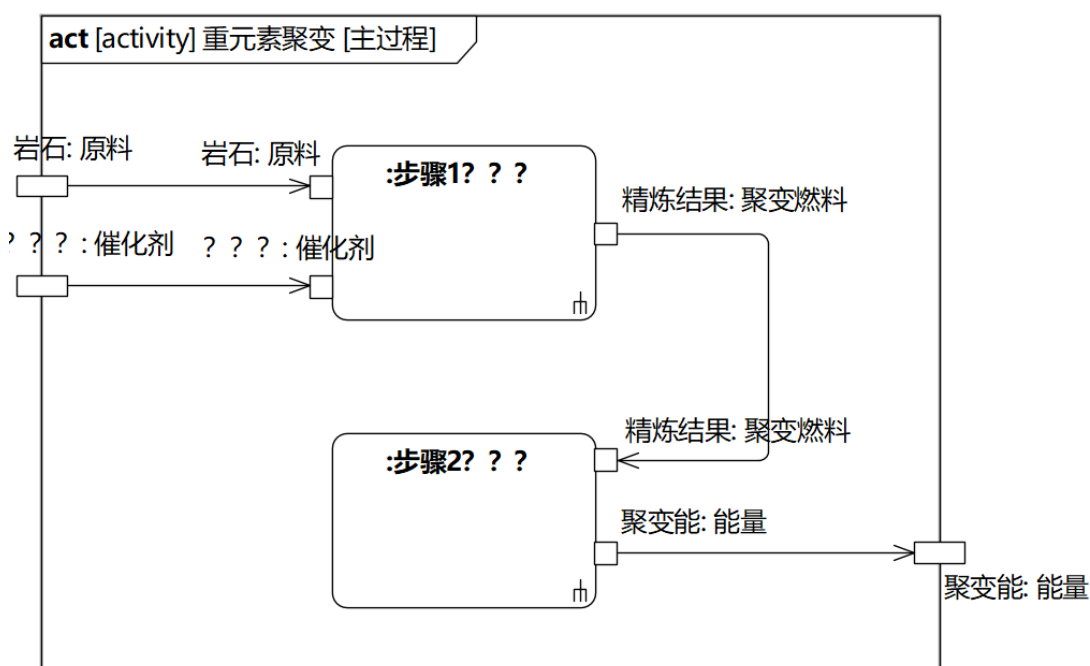


图 1-17 重元素聚变过程

打问号的地方，例如聚变过程的各个步骤应该怎么进行，催化剂是

哪些，需要什么样的反应设备（1000 万°C的高温啊!），这些难题都需要各个学科的专家去攻克。和这些难题相比，通过 UML 建模构思信息系统的内部结构的难度就不算什么了。

信息系统可以看作是人脑的替代。像上面说的“根据接收到的测量参数值（可能有上万个），计算地球发动机下一步的最佳行动”，这个事情人也可以做到，因为计算的方法就是人指定的嘛，只不过人算得很慢很慢，所以用信息系统来做。

也就是说，我们建造信息系统是要把人脑总结出来的各种知识转移到信息系统中，并用它来取代人脑来做计算。信息系统的一切，包括规则如何表达，数据如何组织，各个部分的耦合、内聚等等，都需要我们一个一个构思。

目前，非信息系统的建模还没有“取代人脑”的压力，SysML 更多是描述和帮助验证。

SysML v2 已于 2025 年 9 月正式发布，最新规范地址：

<https://github.com/Systems-Modeling/SysML-v2-Release>。

SysML v2 有很大的变革，特别是大刀阔斧地清理术语。UML 虽然把各种软件建模“方言”统一成“普通话”，但仍然做了一些妥协，留有各种冗余的尾巴。衍生自 UML 的 SysML 也不可避免存在冗余的问题。SysML 把非必要的术语全部清理（和领域驱动设计圈子热衷造词形成鲜明对比），这是非常值得期待的。

因为 UML 和 SysML 两者非常相似，学习了一个相当于学习了另

一个。如果你的工作只是开发信息系统，那么学习 UML 就可以。如果你的工作确实涉及到开发信息系统和非信息系统，可以直接学习 SysML。在碰到不好用 SysML 建模的问题时，再适当引入 SysML 所没有的 UML 元素。

一条双赢的路线是：SysML 发展到足以覆盖现在的 UML 后，吞并 UML，然后把自己改名为 UML——比 UML 还统一的统一建模语言。商业领域就有类似例子，2005 年，SBC 电信以 160 亿美元收购 AT&T，然后把自己更名为 AT&T。

1.3.4 UML 建模工具和文本 UML 建模

1.3.4.1 UML 建模工具的统计

从 2000 年开始，UMLChina 就在网站上提供 UML 建模工具的统计。最开始是链接到 objectsbydesign，我们负责帮 objectsbydesign 把它的统计文章翻译成中文，现在仍然可以访问，地址为 http://www.objectsbydesign.com/tools/modeling_tools_cn.html。



图 1-18 objectsbydesign.com 网页



图 1-19 2000 年和 objectsbydesign 合作的邮件

★Stuart 当时为此翻译慷慨地支付了我\$1000。现在，翻译一本《分析模式》，估计最终得到的稿酬也比这个多不了多少。

考虑到 objectsbydesign.com 的统计更新频率较慢以及信息覆盖面不够，2002 年，UMLChina 开始自行制作 UML 建模工具一览表，不定期发布。

图 1-20 是一份以 pdf 格式保存的 2003 年 6 月的一览表，共 31 页，摘自《非程序员》电子杂志。该文件的下载地址：
umlchina.com/tools/umltools200306.pdf。

UML 相关工具一览（2003 年 6 月版）：A-H

整理
吴昊 查看评论

两年以前，《非程序员》第 2 期曾经刊登了“UML 相关产品价格”一文。两年过去了，UML 相关工具的发展又如何呢？以下总结了全世界的各种 UML 相关工具，都放上来供大家选择。工具的品种相对于两年前多了许多，总共有一百多种，所以我们以连载的方式刊登。按工具名称字母排序，本期刊登 A-H。

工具(最新版本)	厂商	试用允许	UML 支持							支持代码环境	XMI	平台	备注
			用例图	类图	状态图	活动图	顺序图	协作图	构件图	部署图			
4Keeps v4.2	4Keeps, Inc.  http://www.4keepssoftware.com/4Keepsdemo.htm	30 天试用	✓	✗								Windows	Visio5.0、2000 的插件
Advantage Joe 3.0	Computer Associates  Computer Associates® http://www3.ca.com/Solutions/Product.asp?id=257									Java		Java	就是以前的 COOL:Joe。针对 UML 构件图的 EJB 工具。

图 1-20 2003 年 6 月统计的 UML 工具一览表

图 1-20 的文件共列了 130 款 UML 建模工具，根据我的记忆，这个一览表上收集的工具最多时达 168 款。目前，UML 发布类似信息的主要渠道改成了 UMLChina 公众号，umlchina.com/url/tools.html，如图 1-21。

Rhapsody、drawio: UML建模工具2026年1月到3月更新（1-10）

原创 潘加宇 UMLChina 2026年3月27日 13:09
北京 7人

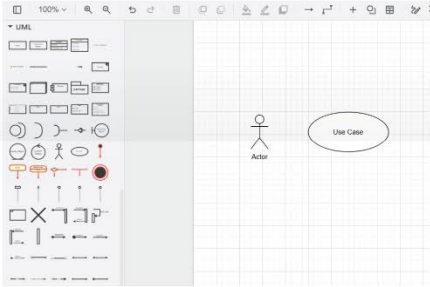
DDD领域驱动设计批评文集

做强化自测题获得“软件方法建模师”称号

《软件方法》各章合集

工具最新版本: drawio-desktop 29.6.6

更新时间: 2026年3月26日



工具简介

开源绘图工具，用Electron编写，跨平台，支持UML。桌面版和在线版现在版本号已统一。在线版: <https://app.diagrams.net/>

平台: Chrome OS、Linux、MacOS、Windows

获得地址

<https://github.com/jgraph/drawio-desktop/releases>

工具最新版本: Visual Paradigm 18.0

更新时间: 2026年3月22日



工具简介

18.0版本已经宣称AI驱动。很用心的UML建模工具。支持编写用例规约。支持文本分析和CRC卡。支持自定义形状或导入Visio的图形。Java代码转序

图 1-21 UMLChina 公众号发布的工具更新统计

★以上统计仅针对涉及 UML 建模的工具，其他建模表示法也会有相应工具，不在统计范围之内。

★感兴趣的读者可以搜索“国产 MBSE”，看看能搜到多少国产建模工具。

1.3.4.2 本书使用的 UML 建模工具

本书中的模型图，如果是我为了讲解知识而画的，用的建模工具主要是 Enterprise Architect，有的图会用到 Rhapsody 和 CATIAMagic。如果是截取其他人提供的模型图，可能还会涉及其他工具。

以上提到的工具仅是我个人的选择，本书不详细评价各款 UML 建模工具。以上这三款工具目前都是收费的，从常理看来，收费工具的体验比免费工具好。如果您基于各种考虑选择使用其他的建模工具，甚至是在纸上手绘，本书的绝大部分建模知识依然适用。

1.3.4.3 文本 UML 建模

近年来，文本建模 (Textual Modeling，营销用语有 Diagrams as Code 等) 工具是 UML 工具市场中增长最快的细分领域。

如果把“从文本生成 UML 模型”都可以看做文本建模，那么上个世纪的 Rational Rose 和 I-Logix Rhapsody 都已经可以把 Ada、C++、Java、VB 代码转成 UML 类图。我们把定义缩窄，把文本建模定义为“以获得 A、B、C 工作流 UML 模型为目的而提供文本”。

OMG 为 UML 定义了元模型、建模元素的语义和表示法，还定义了 XMI (XML Metadata Interchange，XML 元数据交换) 标准用于

各个工具厂商的不同建模工具之间的模型数据交换。这些标准一直沿用至今。

OMG 曾在 2004 年发布过一个 “UML Human-Usable Textual Notation” (<https://www.omg.org/spec/HUTN/1.0/PDF>)，简称 HUTN，试图提供统一的文本建模标准，但该规范未能继续下去。

以文本方式建模的非官方尝试，在这之前已经开始了。2002 年，Diomidis Spinellis 发布了 UMLGraph (<https://www.spinellis.gr/umlgraph/index.html>)，随后又有 TextUML (始于 2006 年，<https://abstratt.github.io/textuml/>)、WebSequenceDiagrams (始于 2007 年，<https://websequencediagrams.com>) 等。

★ Diomidis Spinellis 的书在国内有中译本的有《代码阅读》、《代码质量》。

★ 始于 2001 年的 UMLet(<https://www.umlet.com>)比较难以归类。

目前最流行的应该是 PlantUML (始于 2009 年，<https://plantuml.com/>)，它是当前各个主流 AI 生成 UML 内容的缺省表示。其他类似文本建模工具还有：yUML、Mermaid、MetaUML、nomnoml、dotuml、ZenUML 等。

现有的文本建模工具，绝大多数是基于 Graphviz (始于 1991 年，<https://graphviz.org>) 和 DOT 语言的二次开发。

文本建模的优点

文本建模能够获得推行，主要原因是它降低了获得的门槛。

(1) web 访问

比起类似于 Enterprise Architect 等点击、选择、拖动图形元素的建模界面，实现编辑文本然后渲染成图形的建模界面实现起来相当简单，何况有了 Graphviz。

构造一款基于 web 的“建模工具”难度降低，使得文本建模工具普遍提供在线编辑器，这对于偶尔需要建模的开发人员来说十分便利，他们不需要额外安装软件，目前还都是免费的！

(2) 和程序员常用的工具集成

文本建模时，模型的内容相当于另一门语言的代码，在编码环境、版本控制工具上的处理是一样的。这也使得文本建模在程序员群体中的曝光率大大增加。

★以上只是说“容易获得”，并没有说“容易使用”。比起 Enterprise Architect 等工具“所见即所得”的建模操作，编辑文字并不利于大脑的思考。这一点，在“1.3.2.2 文本 vs. 图形”中已提到。

文本建模当前存在的问题

(1) 绘图而不是建模

目前的文本建模工具所做的其实是“绘图”：建模人员输入文本，工具把文本渲染成形状符合 UML 规范要求的图形，各张图（或各段文本）之间是独立的。

这样做实际上并不算建模。就像机械制图中的主视图、俯视图、侧

视图都是描述同一个六角螺母一样，建模时，图只是从某个视角对某些模型元素的观察，不同图上的内容是有对应关系的。

注意，上面只是说目前文本建模工具存在这个问题，**并没有说这个问题是由文本建模引起的**。如果工具没有封装 UML 元模型的逻辑，只有图形处理的逻辑，都可能存在类似问题。

也就是说，界面是文本输入还是点击拖拽，并不是判断一个工具本质上是绘图工具还是建模工具的标准。A 工具可能只有文本输入，文本输出，也不渲染图形，但它有可能是建模工具，因为它封装了 UML 元模型的逻辑；反之，B 工具界面操作方便，图形在保持 UML 规范所要求形状的同时，用尽美工手段打磨得十分高大上，但它有可能仅仅是一款绘图工具。

如果本质上是当成图形来处理，还时不时会出现另一个问题：复杂图形（例如复杂状态机）的处理错误：不提供支持，或者无法渲染，或者得到的图形看起来违反语义。

(2) 不统一

各个文本建模工具都有自己的语法，把 PlantUML 的脚本放到 Mermaid 工具中渲染，是得不到图形的。难道要再搞一个像 XMI 一样的交换标准？

过度吹捧文本建模的危险

上面讨论了文本建模当前的优缺点，建模人员可以在适合的场合下使用文本建模。**如果有人（例如伪创新圈子）过度吹捧某个文本建模工**

具“敏捷”，很有可能是把它当成遮羞布了。学习了本书的内容后，你可以用《软件方法》提供的照妖镜去照一照他产出的各种工件，到处都是脓包的可能性非常大。

文本建模的将来

新版本的 SysML 同时支持文本表示法和图形表示法，完全可以覆盖现有的各种“文本建模”，而且足够严谨。这是一个努力的方向，即使 SysML v2 没有达到统一的目的，SysML v3 再努力。

本书使用的文本建模

本书所使用的关键模型，除了 EA 版本之外，还会提供相应的 PlantUML 格式的文本。

1.4 应用 UML 的建模 workflow

1.4.1 UML 工具箱

UML 2.5 包含的图形一共有 14 种，如图 1-22 所示的树上的叶结点。

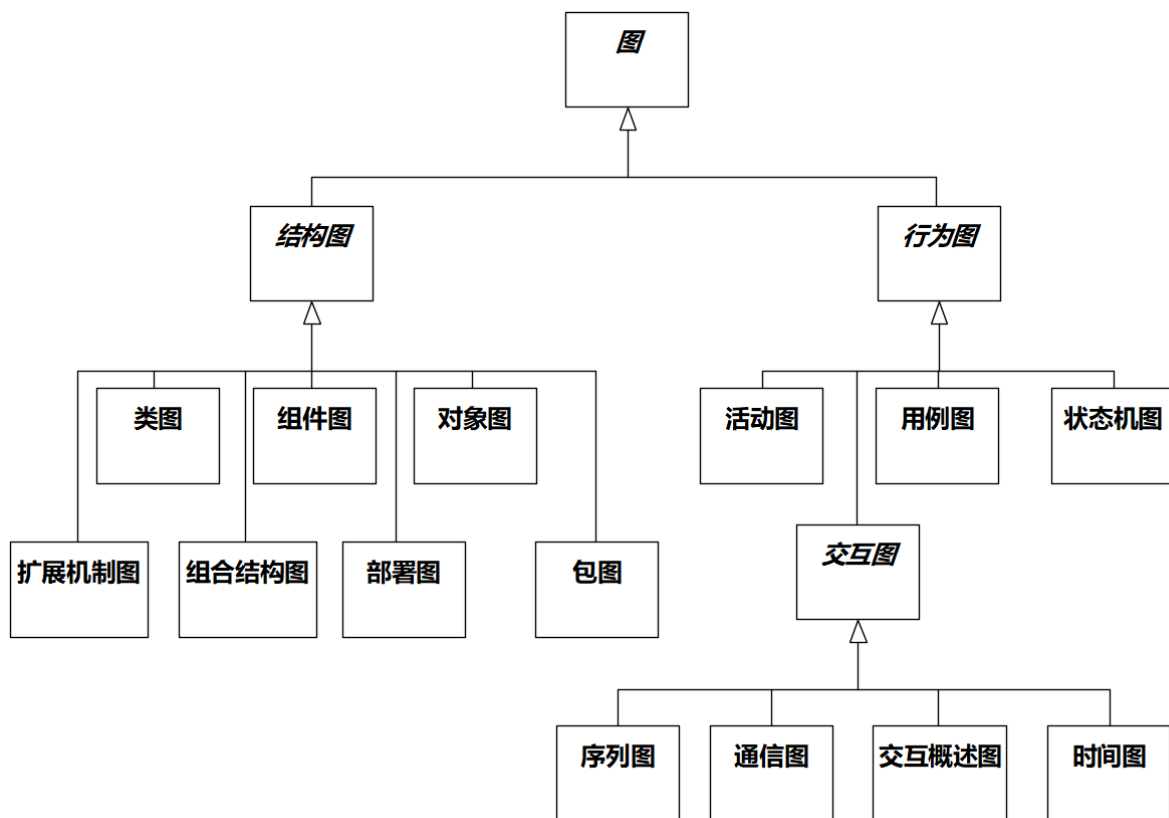


图 1-22 UML 图形（根据 UML2.5 规范绘制）

经常会有人有意无意地误导，让人以为使用 UML 建模需要画这么多图，顺势攻击“UML 笨重”。其实，应该把 UML 看作一个工具箱。工具箱里面有各式各样的工具，建模人员只需要根据当前所开发系统的特点，从这个工具箱中选择合适的工具就可以，并不需要“完整地”使用 UML。

这和编程语言类似。很多人说“我用 Java”，其实只是用 Java 的一小部分，而且很长时间内也只会用这一小部分。

经常有学员问：潘老师，能不能给一个案例，完完整整地实施了整套 UML？这是误解。这样的案例不应该有。这相当于问：有没有一本经典的小说，把字典里所有的单词都用上了？

有一些建模工具自带的案例模型会造成误解,一个模型里展示十八般武艺,把所有的 UML 图都给用上了。这是工具厂商出于展示其工具建模能力的目的而提供的,真实项目中不要这样去模仿。

我用类图表示建模 workflow 相关概念如图 1-23。

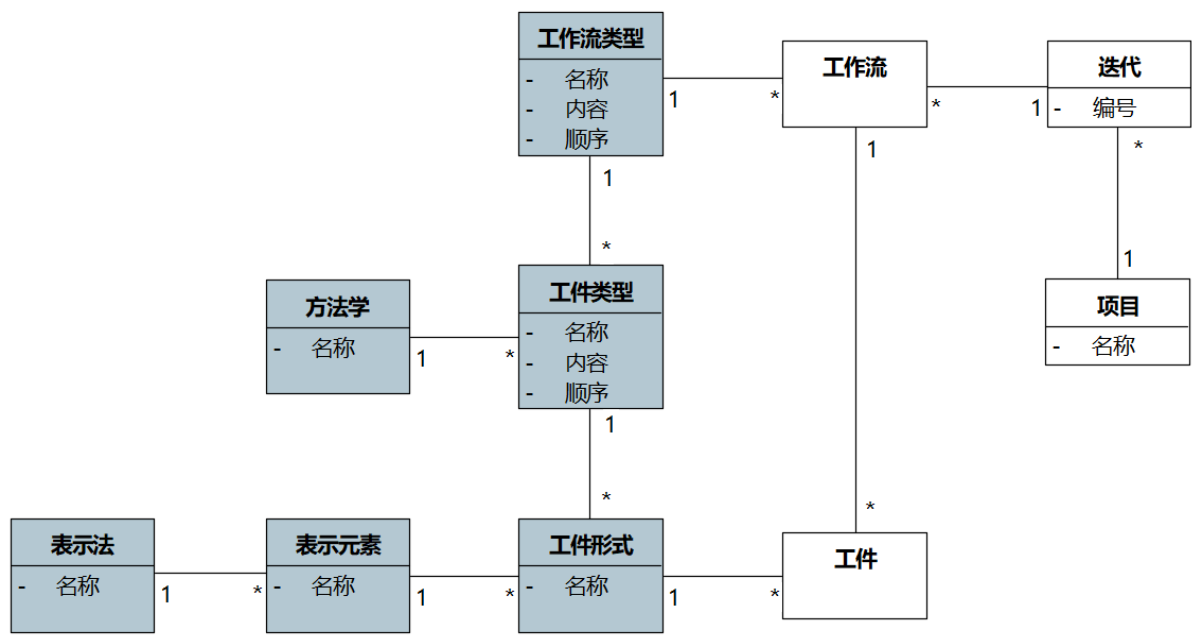


图 1-23 建模 workflow 相关概念

图 1-23 左侧灰色部分定义了“游戏规则”，右侧则是在“游戏规则”上“玩游戏”的结果。

我用对象图展示本书 202505 版本之前推荐的“游戏规则”如图 1-24，以帮助读者更好地理解相关概念。

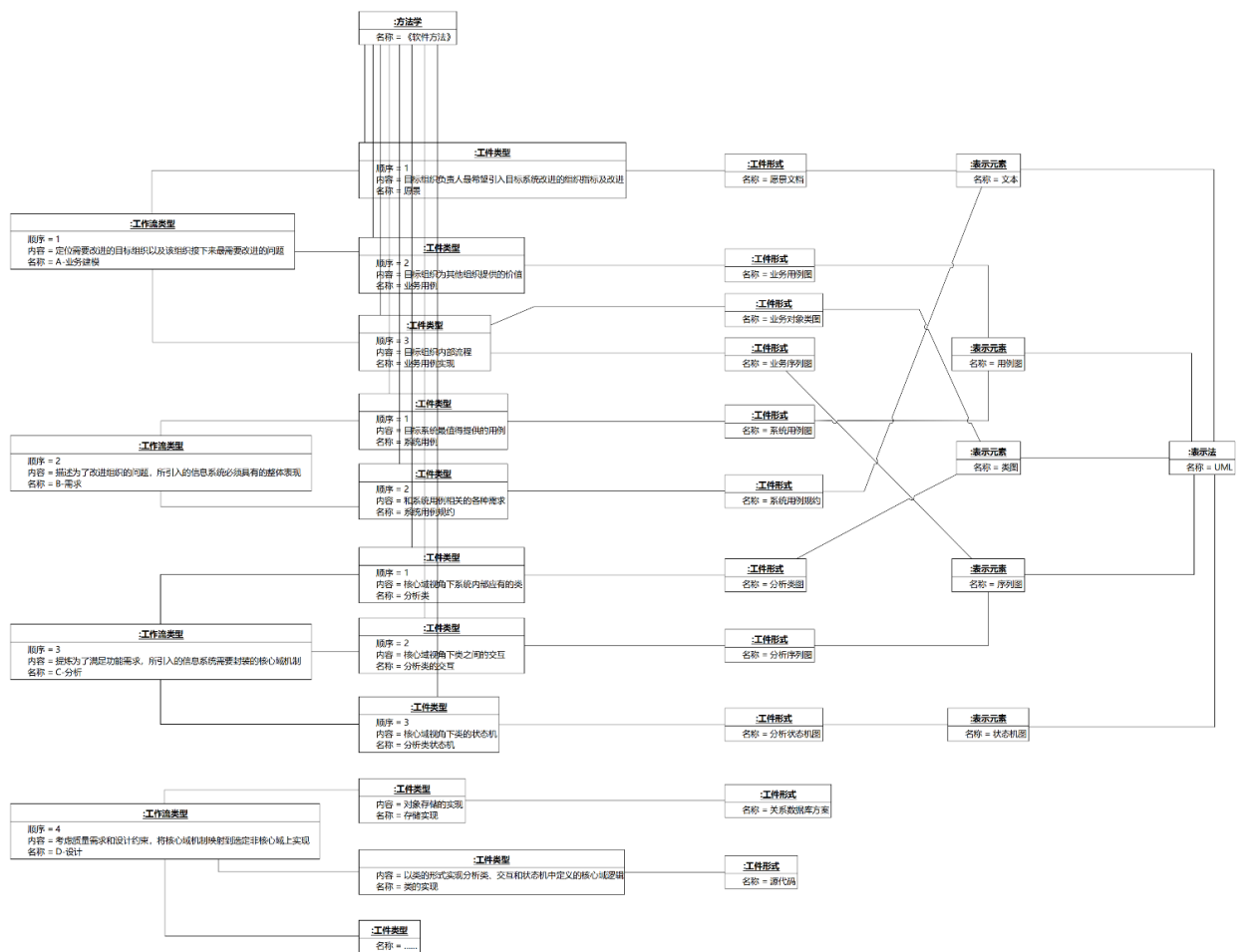


图 1-24 建模 workflow 相关概念的对象图 (图形较大, 点开看或访问原图 umlchina.com/book/1pics/fig1_024.png)

图 1-24 中, 除了最左侧的“工作流类型”之外, 都是可以变化的。

例如, 如果决定将系统用例规约改为用活动图表达, 我们只需要把图 1-24 中的工件形式“系统用例规约”和表示元素“文本”之间的链接改为工件形式“系统用例规约”和表示元素“活动图”之间的链接, 如图 1-25。

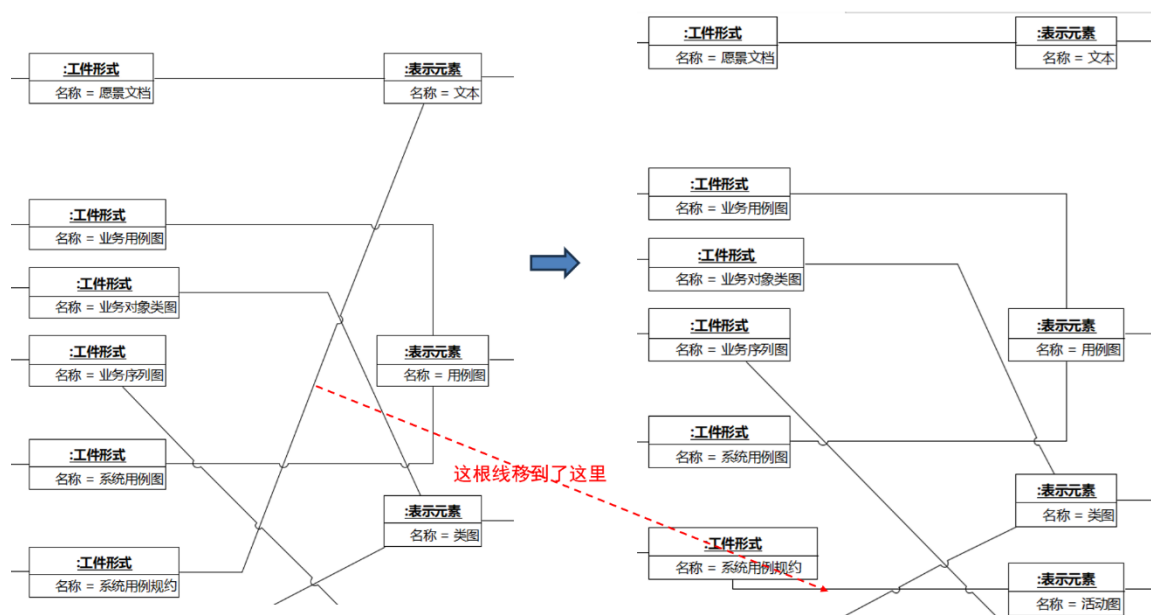


图 1-25 系统用例规约改为用活动图表达

从图 1-24 到图 1-25 是更换“系统用例规约”这个工件的表示元素，从“文本”更换到“活动图”。

★也可以更换“表示法”。例如，把 UML 换成其他表示法或自创的表示法，那么图 1-24 左起 3 到 5 列都要修改。

★也可以更换“方法学”。例如，受“少林功夫+唱歌跳舞”启发，创造出“浑元形意太极+革命性创造划时代洞见领域驱动设计敏捷精益方法学”，但仍然使用 UML 表示法，那么图 1-24 左起 2、3 列需要修改。

从图 1-24 和图 1-25 可以看出，本书推荐使用的 UML 图形有 5 个：用例图、类图、序列图、活动图 and 状态机图。图 1-26 列出了其他可选的用法。

workflow	用例	类	组件	对象	部署	组合结构	包	扩展机制	序列	通信	状态机	活动	时间	交互概述	文本
业务建模	●	●					√		●			√		√	●
需求	●						√		√		√	●			√
分析		●		√		√	√		●	√	●	√	√		√
设计		√	√	√	√	√	√		√	√	√	√	√		●

图 1-26 可选和推荐的建模元素用法（●表示推荐使用，√表示可以使用）

从图 1-26 可以看出，设计工作流的建模，推荐做法是不用 UML 表达，而是用相应实现平台的表示法表达，即所谓的“源代码”（目前大多是文本形式）。考虑了具体平台实现的类图、序列图、状态机图等，其中包含的信息和“源代码”差不多，没有必要花费精力去画设计工作流的 UML 图再编码，因为这样做没有带来任何增值。

更合理的做法是，做好分析工作流，把领域逻辑放进分析模型中，然后再结合实现平台的特点，定制从分析到设计的有规律的映射套路，通过建模工具或者人力搬砖把分析按照套路映射到设计。

这时，即使要画设计的类图、序列图等，也只需要挑选典型的类、典型的用例来展示映射的套路，不需要把分析模型的内容都结合实现平台画一遍。

像 Rhapsody（当前的全名是 IBM Engineering Systems Design Rhapsody）这样的建模工具，可以和各种开发环境集成，配置好后就可以通过正向工程从类图、状态机图生成可执行的代码。开发人员甚至

可以做到只需要编辑和调试 UML 模型，不需要在编码环境中编辑和调试。图 1-27 是用 Rhapsody 工具绘制的某个模型（洗碗机）的运行时状态机图，粉红色标出了当前的状态。类图和状态机图的背后有真实的 C++ 代码。

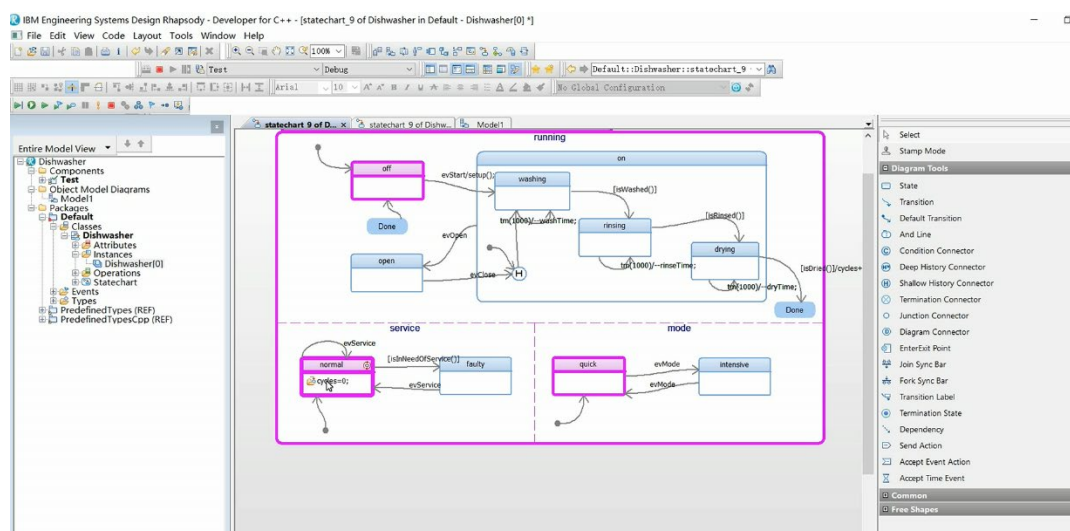


图 1-27 Rhapsody 下的运行时状态机图

当然，Rhapsody 支持的平台和语言只是少数，而且目前很多建模工具在分析映射设计方面达不到类似 Rhapsody 的水平，但这并不妨碍我们针对所选择的实现平台来定制映射套路，然后通过人力搬砖来达到类似目的。在有 AI 辅助的情况下，通过例子不断训练 AI，直到它掌握套路，可以代替大部分原来由人力搬砖的工作。

同样，如果需要设计工作流的 UML 图来搞形式主义充场面，可以通过建模工具对源代码或数据库做逆向工程，生成设计工作流的各种 UML 图。如图 1-28，就是用建模工具 UModel 对某个 C# 类的某个操作做逆向工程生成的序列图。可以看到，这张序列图涉及到很多类的协作。图的右侧放大了一个小片段，勉强让读者能够阅读。

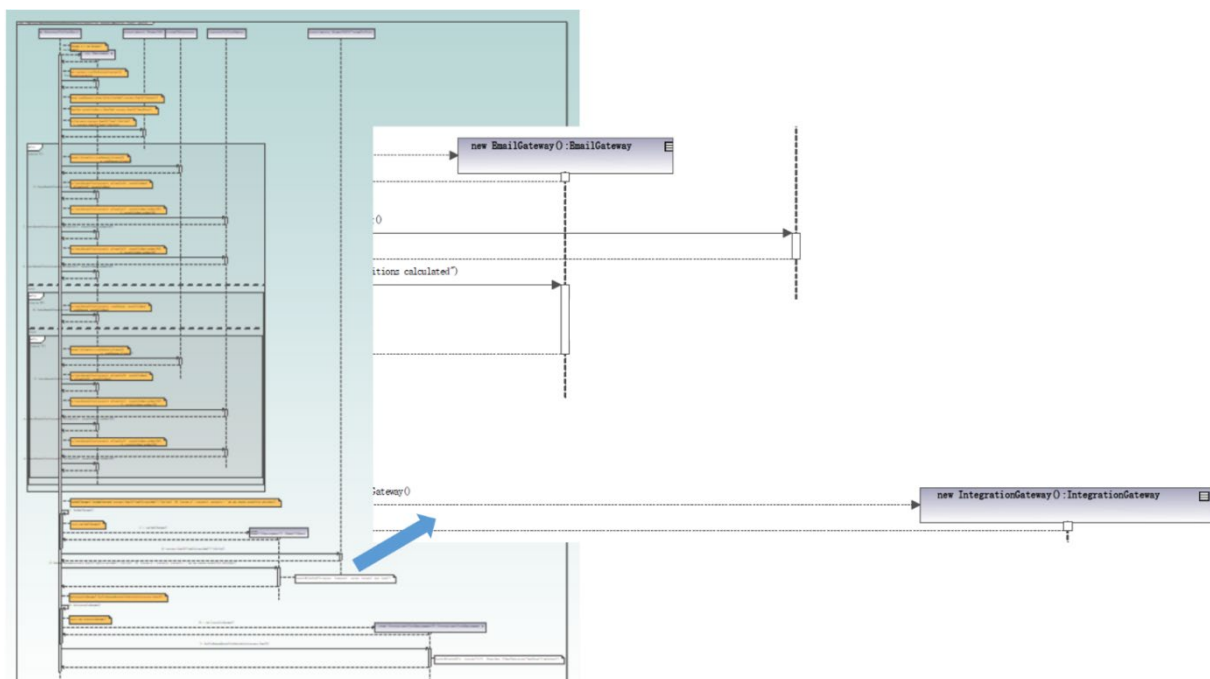


图 1-28 从代码逆向工程得到的 UML 序列图

建模过程中，我们写的每一个字，画的每一张图，都应该能带来增值，否则就没有必要写它或者画它。这一点，在 AI 生成内容已经极其廉价的今天尤为重要。本书后面还会不断提到。

以上提到的定制映射套路、正向逆向工程等内容，本书在设计工作流的章节会再详述。

1.4.2 本书方法学推荐的建模步骤

我用活动图进一步表示本书方法学推荐的建模步骤如图 1-29，让读者看到这些工件是怎样出现的。本书的内容就是按照图 1-29 的顺序讲述。

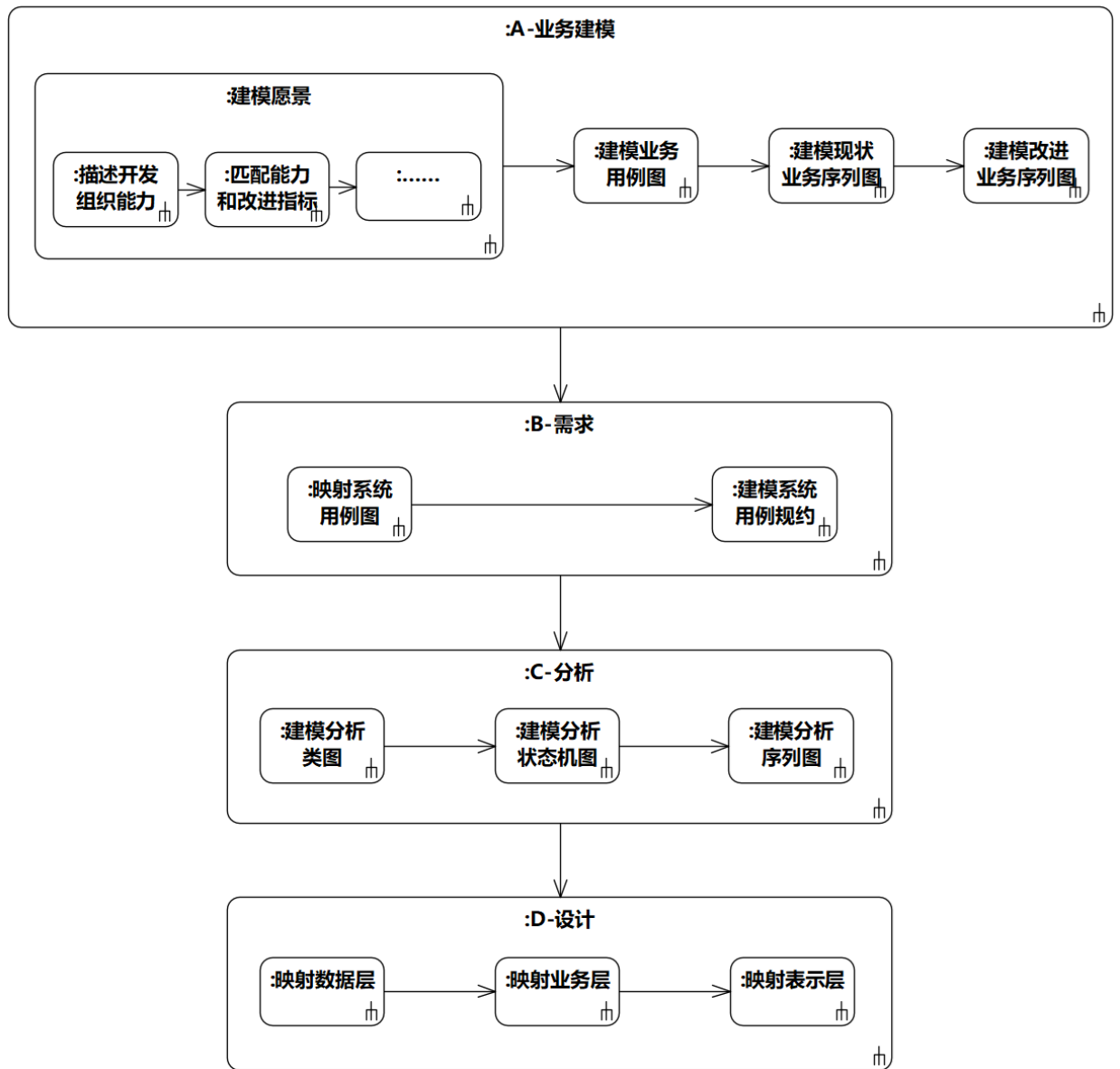


图 1-29 使用 UML 的 ABCD 建模 workflow 步骤

图 1-29 的各个步骤，都可以用 AI 辅助。我们把 AI 当成一名学习能力极强的学生，用本书的方法学引导它，利用它掌握的资源帮助我们提高每个步骤所产出工件的质量。

例如，图 1-29 左上角的“建模愿景”在 AI 辅助下的展开，可以如图 1-30：

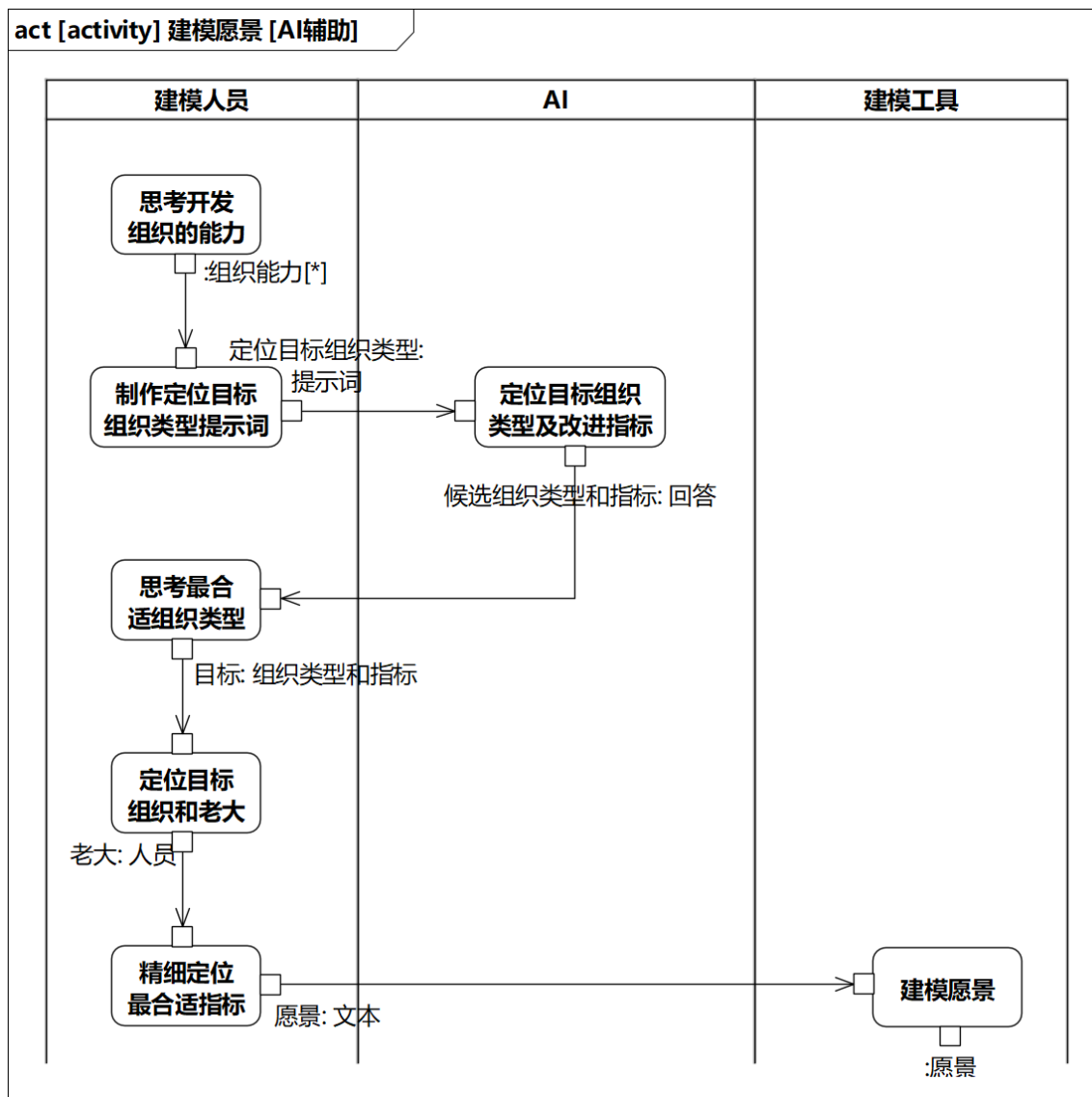


图 1-30 AI 辅助下建模愿景

1.4.3 关于“源代码就是设计”的歪曲和真相

1.4.3.1 Jack Reeves 的文章

1992年,Jack Reeves在“C++ Journal”上发表了一篇名为“What Is Software Design (什么是软件设计)”的文章。

我把 Jack Reeves 文章中的观点列出如下，并附上我的评论。

(1) 根据他的观察，软件开发生命周期中，唯一满足工程标准的

软件文档是源代码清单。

——这是正常的。当时（1992 年），除了源代码所使用的编程语言之外，软件开发生命周期中其他产出很难谈得上有什么严格的语法和语义规则。

（2）软件开发的一切都是设计过程的一部分。编码是设计，测试和调试是设计，平时我们称为“软件设计”的东西也是设计。

——这是要求从“设计”的高度来看编码，不是说代码能编译运行就行。

（3）C++是很有表达力的软件设计语言。

——要从“设计”的高度来看编码，编程语言要有好的表达力，例如 C++。毕竟是发表在 C++ Journal 上的文章嘛。

整体来看，Jack Reeves 的文章没有太偏激的观点。即使存在问题，也是前文提到的普遍缺少 A、B 工作流知识的问题。

1.4.3.2 Robert C. Martin 的修饰

"What Is Software Design" 一开始并没有广为流传。

2002 年，Robert C. Martin 在他的书 "Agile Software Development, Principles, Patterns, and Practices" 第 1 版中附上了这篇文章，并在前面加了个大标题 "The Source Code Is the Design (源代码是设计)"。

这是 Robert C. Martin 的第一个“小心思”。

Robert C. Martin 这本书的书名也包含着 “**小心思**”。“Agile Software Development, Principles, Patterns, and Practices” 主要讲的是面向对象设计的一些方法（原理、原则和模式），这些方法并非 Robert C. Martin 首先提出，和 “敏捷” 提倡的 “过程” 也没有必然关系，但 Robert C. Martin 把它们称为 “敏捷**”，很容易让开发人员误解，这些东西是 “敏捷” 人士提出来的。

Section 2	Agile Design	85
Chapter 7	What Is Agile Design?	87
Chapter 8	SRP: The Single-Responsibility Principle	95
Chapter 9	OCP: The Open–Closed Principle	99
Chapter 10	LSP: The Liskov Substitution Principle	111
Chapter 11	DIP: The Dependency-Inversion Principle	127
Chapter 12	ISP: The Interface-Segregation Principle	135

图 1-31 “Agile Software Development, Principles, Patterns, and Practices” 的目录

最近一些年的领域驱动设计伪创新走的也是这条路线，背后的操盘者们也是来自同一个圈子。更多评价见后文提到的 “染色”。

2007 年, Robert C. Martin 在第 2 版 “Agile Principles, Patterns, and Practices in C#” 中，也附上了 Jack Reeves 的文章，但名字改成了 “What Is Software (什么是软件)”

1.4.3.3 中文翻译的误导

Robert C. Martin 书的第 1 版中译本《敏捷软件开发：原则、模式与实践》中，把 “The Source Code Is the Design” 翻译为 “源代码就是设计”。

“是”和“就是”有微妙的差别，“是”表达的是从属关系，“就是”隐含着相等关系。

马铃薯是草本植物：马铃薯 $\in$ 草本植物

马铃薯就是土豆：马铃薯=土豆

源代码是设计：源代码 $\in$ 设计或源代码 $\subset$ 设计

源代码就是设计：源代码=设计

从上面列出的观点（2）可以看出，Reeves 的文章中并没有这个意思。

文章“What Is Software Design” 目前可在以下地址获得：

https://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf

感兴趣的读者可以去下载阅读。

另外，以上只是说 Reeves 的意思被歪曲了，但没有说 Reeves 本来的意思就一定是对的。类似这样的观点，往往都存在前文提到的普遍问题：缺少对 A 和 B 工作流的认识。

关于“source code (源代码, 代码)”，有的开发人员也存在误解。

计算机运行的是二进制指令，源代码是程序员需要理解和编辑的最低形式模型——编辑完这个，后面的事情就可以交给计算机了！

中文翻译“代码”中的“代”字就体现了这个意思，用更方便理解

和编辑的符号来“代替”原来的指令。从这一点看，编码就是一种最低形式的建模工作，源代码也是“模型”。

这个最低形式模型是随着时代的发展不断变化的,如图 1-32 所示。

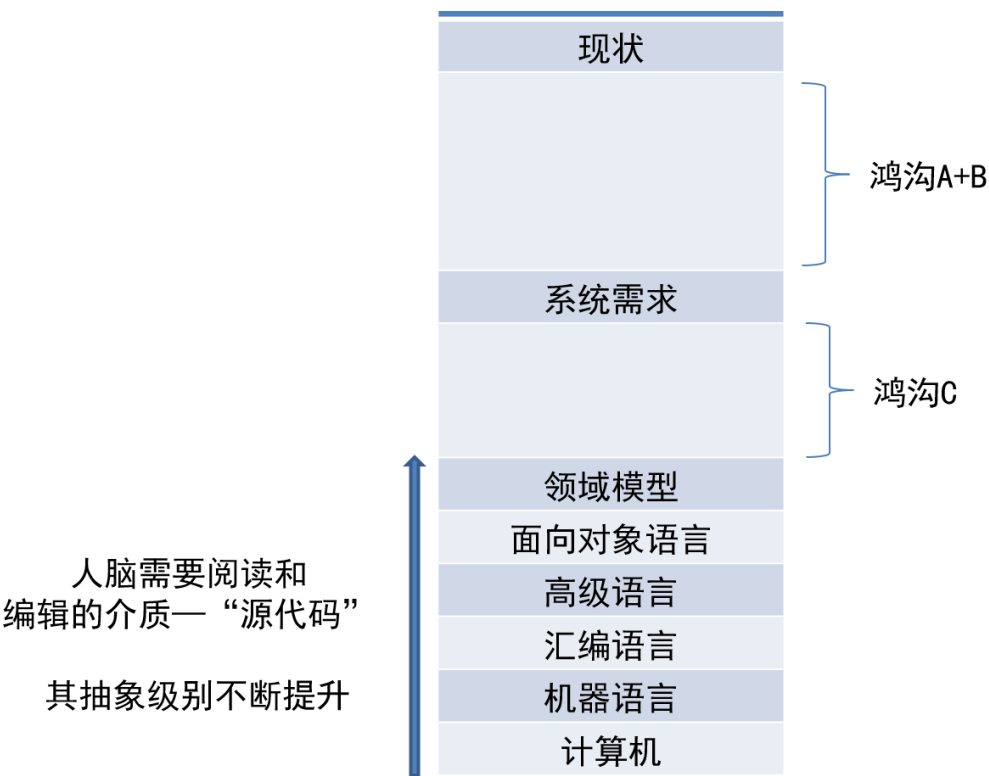


图 1-32 “源代码”的发展历程

最开始，程序员在纸带或卡片上打孔来表达 0 和 1，这可不是“代码”了，就是“码”本身。

后来发现这样太累了，于是发明了一些助记符，这就是汇编语言。

有的开发人员凡尔赛，“这些我不太懂唉，我是做底层的，用 C 编码”，可是 C 语言却被归类为“高级语言”，因为类似 C 这样的语言出现的时候，大多数程序员编辑的是汇编语言，C 相对于汇编来说，当然很高级。

今天的一名企业应用程序员，需要编辑的可能有 HTML、CSS、

JavaScript、Java、配置脚本、SQL 等，这些就是当前常见的“源代码”的形式。

如果人脑只需要编辑 UML 模型就可以实现系统，不需要编辑 Java、C# 等文字形式代码，那么 UML 模型就相当于“源代码”。这是目前可以期待的，也是图 1-32 中向上的箭头目前能到达的。

图 1-32 中的“鸿沟”则表示目前还非常需要人脑（建模人员的大脑）去解决的部分，后面在谈到人工智能对建模工作流的帮助时还会详细讲解。

因此，不存在什么“无代码”、“低代码”，人最终需要编辑的那个东西就是“代码”。

★很多（15+）年前，一位 CTO 向我展示他们的软件，说不用编码就可以为不同的客户定制（当然，变化范围也有限）。我问他怎么做到的，回答“只需要改配置文件里面的各种信息就行！”。其实那个配置文件就是一种“代码”。

★今天（2026 年），如果一个人对当前主流编程语言毫无知识，纯粹靠 Vibe coding 来开发软件，那么和 AI 交互的提示词可以看作源代码；如果他还需要审核 AI 生成的源代码，即使他不亲自修改，只是向 AI 反馈，提示词也不算源代码。

总之，关于“源代码是设计”，不管是 Jack Reeves，还是后来花心思包装的 Robert C. Martin，认识都是不够严谨的，正如 1.2.4.1 思维颠倒所说的，主要问题出在对“设计”的认识上。

1.5 警惕和揭秘伪创新

初中数学里要学习全等三角形、相似三角形、SSS、SAS……，到了高中以后学了正弦定理、余弦定理等解三角形的知识……**就不会再回去用初中的方法解题了。**

但是，不是所有人都能学会高中的知识，比如说张三。

张三可能会这样解释：

“我这个人能力比较弱，只能掌握全等三角形、相似三角形的方法。”

这样的说法没有问题。

张三还可能会这样解释：

“这个题目比较简单，用全等三角形、相似三角形的方法做足够了，而且这样更方便广大人民群众理解。”

这样的说法也可以。不过，竞争对手不是傻子，**市场中哪里有什么“简单题目”！能带来利润的题目都很复杂。**

但是，张三如果这样说：

“全等三角形、相似三角形的知识比高中三角函数的知识更深刻。”

这是自欺欺人。

更要警惕的是，有一个李四，也许和张三一样没有掌握高中方法，也许掌握了高中方法但是为了忽悠张三们，偷偷把“全等三角形”改名为“叠合三角形”，然后和张三宣传：

“我发明了“叠合三角形”新方法，比高中的三角函数有用，三角函数过时了。”

这就是可恶了。

前文已经提到过伪创新，此处，我们再来详细谈一谈。

1.5.1 你情我愿的伪创新

前面 20-30 年间，互联网和信息行业的大发展，使得大量没有受过应有训练的人员加入并从事软件开发工作。他们中的一部分人并没有“补课”的意愿和能力，而是致力于寻找可以让他们逃避辛苦学习的伪创新。他们是“伪创新”的买家。

某些开发团队的工作实际上是在装模作样，浪费时间在那里废话刷工作量。在“猪都会飞”的时期，特别是在开发团队的能力不是关键竞争要素的领域，废话刷工作量的危害没那么明显。而开发团队为了突出自己的贡献，想表明所在组织取得的成功来源于本团队超人一等的能力和努力，迫切需要拉上某些“方法”作为面子和证据。他们是“伪创新”的买家。

有买家就有卖家，于是，国外国内就有人炮制出各种投资少，见效快，产量大、门槛低、仪式感十足的伪创新，组成圈子向开发团队宣传。

开发人员和开发团队一接触，欢天喜地连称“受用”，更加心安理得地假装努力，骗客户，骗领导，骗自己。

前些年，伪创新以“敏捷”为名，近年来，伪创新以“DDD（领域驱动设计）”为名。细心观察可以发现，背后的主要推手是同一批人。

1.5.2 不学有术

严谨而系统化地研究某个领域中前人的成果,清楚前人曾经想过什么,做过什么,哪些成功了,哪些失败了,才能提出真正的问题并解决问题——这是真正的创新。

伪创新圈子的“创新”却来得非常容易,可以说是“不学有术”——不学习却能大量输出。

他们不去认真研究前人的成果,而是基于自己碎片化的阅读和经验,把一些朦胧感悟写出来,然后造一个新词来命名,就得到“创新”了。

★有的网红甚至是这样“创新”的:我和公司同事某某(也是网红)讨论了什么什么,就悟到了某某创新。

在伪创新圈子的文字里,常会见到“我突然想到”、“我发现”、“我悟到了”等字眼。

如果仅仅是发表自己的体会或感悟,即使内容是错误的,也是个人自由。伪创新圈子的问题在于:他们硬要把自己的体会包装成“创新”,然后向大家布道。

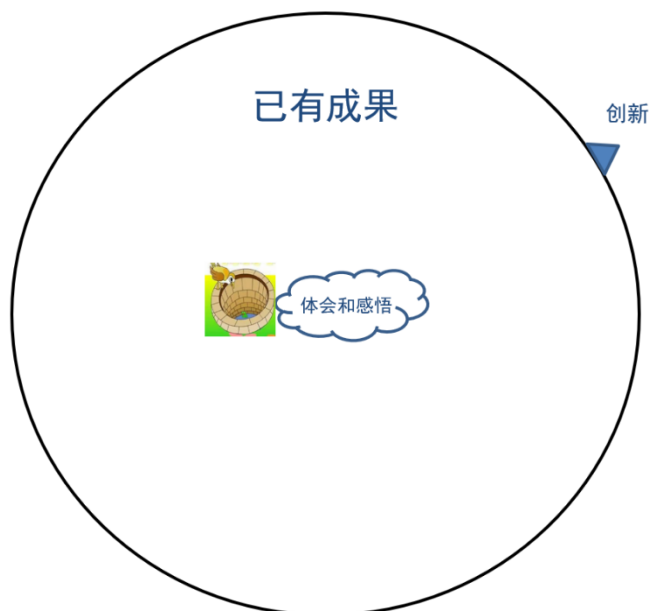


图 1-33 体会和创新的区别

天下没有免费的午餐！“不学”却“有术”的伪创新，它的内容符合以下特点之一：

(1) 错误

不管是过去的开发团队，还是现在的开发团队，用上了不但没用反而有害。错误的内容可谓是“百花齐放”，什么样的都有。

(2) 过时

悟出来的内容，其实之前已经存在而且在一定时期内发挥了作用，但现在已经被更好的知识取代。

例如，某 DDD 实践者分享了自己创造的架构表示法，仔细一看，不就是 1970 年代的数据流图吗？现在在需要画数据流图的地方，使用 UML 或 SysML 活动图是更好的选择。

(3) 拙劣的模仿

例如，某发明家急于“创新”，模仿现在仍在发挥作用的方法学自己搞了一套，只不过各种名称换成了自己的“造词”。由于不下功夫学习，囫圇吞枣，“创新”的“方法学”比起现有的成熟方法学来，到处是倒退和漏洞。

学霸做了一份试卷，比较难，估计拿不到满分。学渣自告奋勇，硬是要帮学霸改一下以提高分数，你猜会是什么结果？

该掌握的知识没有掌握，就硬着头皮“创新”，后果可想而知。

上面所说的不只是符合软件开发的伪创新，网络上的各种“数学民科”、“物理民科”的“数学创新”、“物理创新”也是非常符合的。

如果这些“民科”能花时间去认真学习大学（本来想说高中，但听说中学要求的学习深度在逐年下降）的数学和物理——就当是卧底，带着批判和找茬的目的去学——学习到能独立把大部分习题做对的程度。

如果他们真的能做到这一步，估计也就不会再相信之前自己瞎掰的那些东西了。

可是有多少“民科”会这样做呢？如果他有如此认真学习的兴趣和毅力，之前他也不会“不学有术”地频频“创新”了。

我也曾经和软件开发的伪创新圈子说过多次，就算是带着批判和找茬的目的，把我的书看一看，把我出的几百道题做对了。

同上，要是真的把题做对了，他也就不会再去相信那些伪创新了，甚至会为曾经相信那些东西而羞愧。

可惜，没有人能做到。有的人看两页书，题也不做，就啪啪啪发一堆感悟——然后，一直停留在那里。

最后要说明一下：

“民科”说的是做事的态度和方法，并不绑定具体身份——例如定义官方研究机构之外的研究者一律为“民科”。

不少真正的创新来自企业。特别是 IT 业，一些创新甚至来自非正式团体或个人。不过，官方研究机构的人员筛选，使得伪创新的概率会小一些，至少知道写东西之前要先做研究。

而企业特别是互联网企业在“猪都会飞”的膨胀期，吸纳了各种各样没有严谨态度的人员——所以，大家可以看到一些头部互联网企业会加入很多伪创新的推波助澜中，特别是员工喜欢到处“布道”的企业。

当然，也有著名科学家年轻时真正在研究科学，进入老年之后大脑退化，不能也不愿沿着科学的道路继续探索，同时又面临死亡的无解难题，于是转为宣传宗教甚至宣传迷信。

1.5.3 造词

伪创新圈子很擅长造词，恨不得每个用词都能当演讲题目。

如果人们得知一个东西曾经存在过，那么当这个东西再次被拿出来宣传时，人们会对宣传持有较多的理性，“这东西如果真的这么厉害，那之前怎么……”，宣传的人也会收敛，不至于那么夸张。如果起一个新名字，就会给人一种“全新”的感觉，人们可能就会给一个机会，毕竟是“新”的，没准人家真的有这么牛呢。

例如，说青霉素可以治愈肝癌，大众肯定不信，要是真的可以治愈肝癌那么多年不早就验证了嘛；如果把青霉素改个名字叫“K9527- α ”，说可以治愈肝癌，可能就会有患者给它一个机会，买了试一试。

一个机会就够了！大不了这个词不好使了再换个词呗，造词工厂的流水线可是全年开工的！

并不是说青霉素没有价值，杀灭还没有耐药的细菌总是可以的。就算是淀粉做的假药，还可以起到充饥的作用呢。

危害在于，被误导的肝癌患者可能会将宝贵的金钱和时间资源优先用在了“K9527- α ”上，**耽误了获得更好治疗方案的机会。**

1.5.3.1 造词手法 1：换词

造词手法可以是换词，相当于 $X \rightarrow Y$ 。

例如，把“术语表 (Glossary)”换成“通用语言 (Ubiquitous Language)”。

换词是比较粗暴的，下面的做法就比较温和。

1.5.3.2 造词手法 2：微调

造词手法可以是微调，相当于 $mX \rightarrow nX$ 。

微调可以是加减数量，例如，把“六边形架构”的边数减 2，得到“四边形架构”。

微调可以是换某个字，例如，把“彩色建模”改成“四色建模”。

1.5.3.3 造词手法 3：加前缀

造词手法可以是加前缀，而且可以加不止一个，相当于 $X \rightarrow AX$ 、 ABX。

例如，在“愿景”前面加前缀“领域”，得到“领域愿景”；在“事件”前面加前缀“领域”，得到“领域事件”；在“需求”前面加前缀“业务”，得到“业务需求”；在“需求”前面加前缀“用户”，得到“用户需求”。

图 1-34 是知乎上的一篇阅读量还比较大的 DDD(领域驱动设计) 文章，先不说“领域模型.....反映需求的本质”的说法是错误的，光是看在“需求”前面累计叠加了三个前缀得到的“领域内用户业务需求”就叹为观止了。

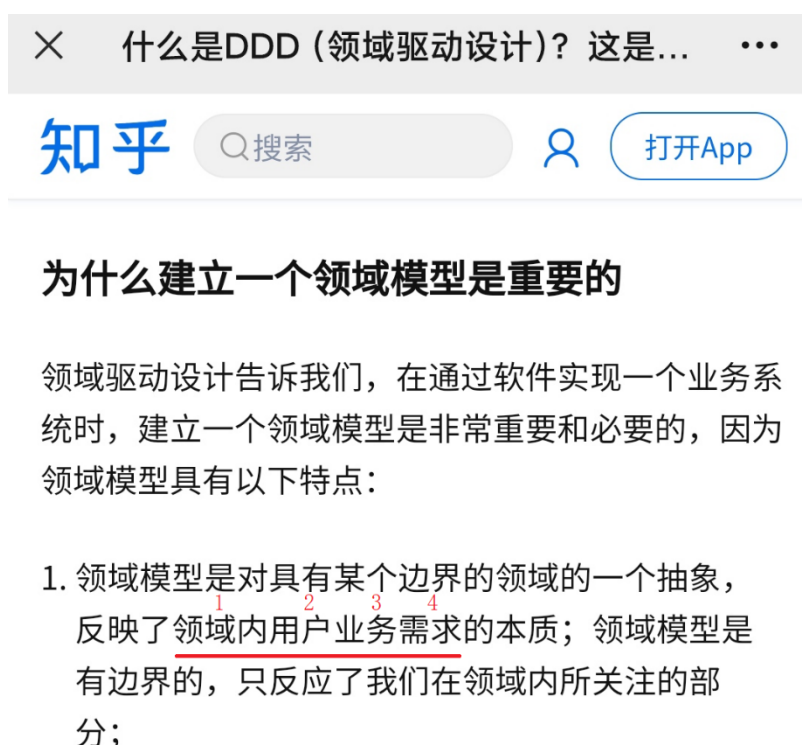


图 1-34 知乎上的一篇关于领域驱动设计的文章

1.5.3.4 造词手法 4：加后缀

造词手法可以是加后缀，而且可以加不止一个，相当于 $X \rightarrow XA$ 、 XAB。

例如，在“功能”后面加后缀“架构”，得到“功能架构”；在“功能”后面加后缀“设计”，得到“功能设计”。当然，还可以连起来——“功能架构设计”。

把前缀后缀都利用起来，可以得到“领域用户业务需求功能架构设计”。

显然，在加各种前缀后缀的时候，“发明家”并不真正了解原词和前缀后缀的真正含义，只是为了强调自己对所加前缀后缀的重视——我非常重视领域、业务、用户、架构、设计。

伪创新圈子喜欢用当下的热词作为前缀和后缀，例如现在的“数智化”、“人工智能”等，因为这些热词给了他们造词的理由—— $\times\times$ 时代都到了，不得整一套新方法啊？

有心的读者可以观察造词圈子的文章，看看有没有类似以下模式的表达：

在[某热词]的时代，[某伪创新]提出[某新词]

在[某热词]的形势下，[某伪创新]提出[某新词]

1.5.4 运作

1.5.4.1 偷梁换柱

步骤如下：

(1) 当 X 比较难时，发明 aX，用 a 来装饰 X。a 通常是某个带有褒义的形式或仪式。

(2) 组圈子互吹互捧 aX，把焦点放在 a 上。

(3) 对 X 畏难的“目标客户”闻风而来，大家热烈讨论 a，心照不宣地假装所有人的 X 已经没有问题。

以“大声建模”为例：

(1) 建模比较难，于是发明“大声建模”；

★“大声”是一个形式，这个形式带有褒义：我大声我骄傲，你要是反对“大声建模”，你就是“偷感建模”。不过，以下人士哭晕在厕所：霍金、图灵（口吃）、牛顿+特斯拉+诺贝尔（害怕交际，终身未婚），“大声建模”对他们来说可太难了！

(2) 组圈子互吹互捧“大声建模”，把焦点放在“大声”上。

(3) 对建模畏难的“目标客户（软件开发人员）”闻风而来，大家热烈讨论如何“大声”，心照不宣地假装所有人的“建模”已经没有问题。

这样的手法也可以很隐蔽。我们来看“领域驱动设计”，这个看起来不符合我们上面说的 aX，别慌，我们这样：

(1) 面向对象分析设计比较难，于是把“领域驱动设计”定义为“新一代的面向对象分析设计”；

(2) 组圈子互吹互捧“领域驱动设计是新一代的面向对象分析设计”，把焦点放在“新一代”（或革命性、划时代）上；

(3) 对面向对象分析设计畏难的“目标客户（软件开发人员）”闻风而来，大家热烈讨论“新一代”，心照不宣地假装所有人对“传统的面向对象分析设计”已经熟悉到“见山还是山”，都不屑一谈了。

这样的手法也可以用在其他领域，例如“微笑手术”：

(1) 手术很难，于是发明“微笑手术”。“微笑”是一个形式，这个形式是褒义的，谁能反对“微笑手术”呢？

(2) 组圈子互吹互捧“微笑手术”，把焦点放在“微笑”上。

(3) 对手术畏难的“目标客户（医生）”闻风而来，大家热烈讨论如何“微笑”，心照不宣地假装所有人的“手术”已经没有问题——**董小姐**们的“微笑”可是碾压各位木呆呆医生！

大家可以拿这个思路去观察还有没有类似这种用形式来掩盖内容的“偷梁换柱”。

我以前在文章里也提到过某次“我和小伙伴都惊呆了”的经历：大概是 18 年前（感慨，今年 2026 年，已经 19 年了啊），一个文科出身的编辑上台大谈敏捷开发一个小时。

没有专门学过计算机科学或软件工程方面的知识（应该没有），也没有从事过软件方面的工作，谈“开发”是谈不来的，但可以谈“敏捷

开发”——重点谈“敏捷”嘛，那可太擅长了！

“偷梁换柱”是谈 a 而避谈 X，如果这一招不凑效，必须得谈 X 了，怎么办呢？

1.5.4.2 山寨

(1) 迎合伪创新买家的喜好，有意或无意地模仿已有的 X、Y、Z 方法，发明出相应的 X-、Y-、Z-方法。

★有意：发明者知道 X、Y、Z 的存在，但只是粗浅了解，没有认真研究。

★无意：发明者不知道 X、Y、Z 的存在，费尽心思发明出 X-、Y-、Z-，实际上仅是对已有 X、Y、Z 拙劣模仿。

★买家的喜好：只了解 D 工作流，不了解 ABC 工作流也不愿意学习。

(2) 把谈论范围限制在 X-、Y-、Z-，避免谈及 X、Y、Z。

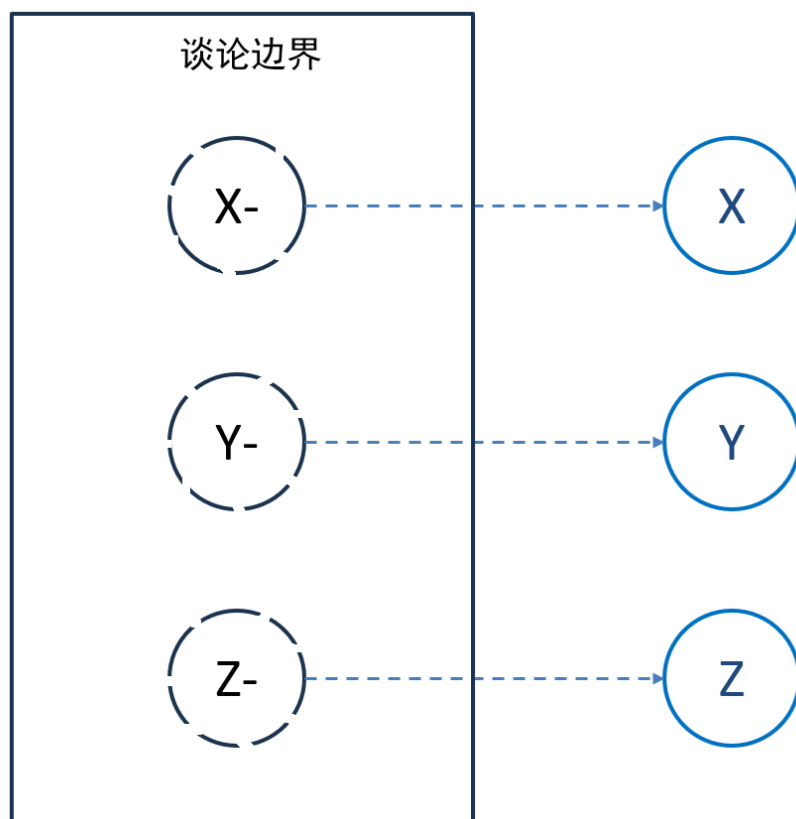


图 1-35 山寨

可能有的读者会下意识认为，哇塞，这不是 GoF 设计模式里面的“适配器 (Adapter)”模式吗？这样一搞，X、Y、Z 能适配不同的目标人群，这是好事啊！

这不是“适配器”。“适配器”是接口不兼容的时候提供“适配”，干活的逻辑还是 X、Y、Z 的逻辑。伪创新则是另搞了一套粗陋或错误的内容，把 X、Y、Z 扔掉了（或者压根没听说过 X、Y、Z）。

也就是说，有可能 X-、Y-、Z-的“接口”和 X、Y、Z 一样，但内容是粗陋或错误的。

1.5.4.3 染色

把自己定义得很抽象，以便把已有成果纳入自己的领地，我把这个叫作“染色”。

“Domain-Driven Design”的作者 Eric Evans 就把领域驱动设计称为“哲学”，他在“Domain-Driven Design”的前言中说：

*Although it has never been formulated clearly, a **philosophy** has emerged as an undercurrent in the object community, a **philosophy** I call domain-driven design.*

不过，一门**哲学**像一股暗流已经在对象社群出现，虽然还从来没有被清晰确切地表述出来。我把这门**哲学**叫作“领域驱动设计”。

把领域驱动设计定位为“哲学”，这个超然的地位使得领域驱动设计更方便地把从前、现在和以后的成果纳入领域驱动设计的领地。

有心的读者可以用“DDD 的”、“领域驱动设计的”、“DDD 提出”、“领域驱动设计提出”等关键词搜索，会得到各种染色的结果，如图 1-36。

通过DDD，结合第一阶段的需求分析，整理收集到的关键信息，并通过DDD的用例分析法、四色建模法或者事件风暴法，初步识别领域服务及相关的界限上下文等关键信息，进而得出初步的领域模型。在此过程中，分析应用相关的业务功能和应用组件，

如果是基于领域，就需要领域专家共同合作，采用领域驱动的方法，根据业务用例进行抽象，达成词根的统一。基于组织的模式可能相对简单一些，而基于消费场景的方式可能无法解决根本的安全性问题，因此建议采用基于领域的方式。

- 第三，DDD 提出的分层架构，有效分离了业务复杂度和技术复杂度，凸显了领域模型，使得领域层的代码和领域模型保持高度一致；

▲ 第四，统一语言非常重要，每个概念有各自的

DDD提出了面向业务领域的软件设计，也就是以业务为驱动的设计

先要瞄准业务需求中的问题。问题空间是解决问题的目标所在，有了问题空间，才能提出解决方案，而解决方案的提出有赖于人们对问题知识的理解。

图 1-36 领域驱动设计的染色

“领域驱动设计”继承的是“敏捷”的风格。“敏捷”可以说是染色的“巨人”了。

2001 年 2 月喊出“敏捷”口号之后，迅速出现一批名为“敏捷××”的书籍。我挑出一些名字相当直白的书（还有很多内容类似的书名字稍委婉）：

Agile Software Development, Alistair Cockburn, 2001

Agile Modeling, Scott Ambler, 2002

Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, 2003

Lean Software Development: An Agile Toolkit, Mary Poppendieck 等, 2003

Agile Management for Software Engineering, David J.

Anderson, 2003

Agile Documentation, Andreas Rüping, 2003

Agile Database Techniques, Scott Ambler, 2003

Agile Software Development, Alan S. Koch, 2004

Agile Project Management, Jim Highsmith, 2004

Agile Project Management, Gary Chin, 2004

Agile Estimating and Planning, Mike Cohn, 2005

Agile Java, Jeff Langr, 2005

Agile Web Development with Rails, David Thomas 等, 2005

Agile Development with ICONIX Process, Doug Rosenberg
等, 2005

Agile Java Development with Spring, Hibernate and Eclipse,
Anil Hemrajani, 2006

Agile Systems with Reusable Patterns of Business
Knowledge, Amit Mitra 等, 2006

Agile Information Systems, Kevin C. Desouza, 2006

Agile Software Construction, John Hunt, 2006

.....

上面所列的这些书籍，大部分有中译本。



图 1-37 敏捷宣言之后的几年内出版的一些书

注意，“敏捷”并不把染色限于项目管理或过程改进，**面向对象、数据库、Java、Web** 等内容也被“敏捷”染了色。

最典型的是 Robert C. Martin 的“Agile Software Development: Principles, Patterns, and Practices”。这本书主要讲述的是面向对象分析设计的一些内容，这些内容非 Robert C. Martin 首先提出，而且和具体过程没有必然关系，但 Robert C. Martin 的做法使很多开发人员产生误解，以为这些内容是敏捷圈子的研究成果。

“Agile Software Development: Principles, Patterns, and Practices”的内容扩展自 Robert C. Martin 于 2000 年在自己网站 objectmentor.com 上发表的“Design Principles and Design Patterns”，Robert C. Martin 在这篇文章中整理了一些他认为比较重要的“原则”和“模式”。

Design Principles and Design Patterns

Robert C. Martin
www.objectmentor.com

What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications¹. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns², packages, components, and classes. It is this level that we will concern ourselves with in this chapter.

图 1-38 “Design Principles and Design Patterns” 截图

★目前，objectmentor.com 已不再有效，在网络上可以下载到 “Design Principles and Design Patterns” 文章的地址是：

https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

“Design Principles and Design Patterns” 的几十页内容中，没有一处提到 “agile”（正常，口号还没喊出来呢），也没有一处提到 “light”、“lightweight”、“process” 或 “methodology”，说明 Robert C. Martin 自己非常清楚这些内容写的是什么。

而在他 2003 年的书中，这些内容摇身一变，成了 “敏捷软件开发原则”。读者可以体会一下，Robert C. Martin 是出于什么样的心思。

这二十多年来，我在和各种开发人员打交道的过程中，就因此备受

困扰。

在圈子的强力宣传之下，开发人员变成井底之蛙，以为所谓的 SOLID 原则就是软件设计的法宝，学会了它，就能搞定软件设计的问题；反过来，如果没搞定，就是因为没理解透这几个原则。

于是，开发团队就来找我给他们讲 SOLID 原则（以及同样出名的 GoF 模式）。碰到这种情况，我也只能如实告知，讲当然没问题，主要就是一个泛化（继承）的应用，车轱辘话产出这么多内容，但除此之外，分析设计还有更难也更有价值的内容值得学习。

读者不妨结合前文内容猜一猜，我这样一说之后，开发团队怎么反应的概率最大？

“敏捷”的“染色”持续到今天。最近几年的各种热点，“敏捷”都有“染色”——敏捷大数据、敏捷机器学习、敏捷人工智能，如图 1-39。



图 1-39 最近几年出版的紧跟热点的“敏捷”书籍

除了圈子众人主动出击四处“染色”之外，当“敏捷”成为政治正确时，一些本来在圈子外的方法学家（甚至还包括 UML 三友之一）眼见口号来势汹汹，于是就带着自己的方法来投靠，宣称“我这个***也是敏捷的!”。

如何让“武德驱动设计”包揽每一年的诺贝尔物理、化学、生理学奖？

(1) 提出“武德驱动设计”的口号，把它定义为“一种强力推动人类文明发展的科学”；

(2) 全球组圈子，散布“武德驱动设计是一种强力推动人类文明发展的科学”的说法；

(3) 称赞乐意投靠的诺贝尔物理、化学、生理学奖获得者符合“武德驱动设计”的精神；

(4) 建立“凡是强力推动人类文明发展的科学就叫武德驱动设计”的思想钢印。

(5) 不认同（至少在口头上）“武德驱动设计”的科学家，将无法获得诺贝尔物理、化学、生理学奖。

你，学会了吗？

1.5.4.4 割裂历史

伪创新会有意无意地忽略掉自己被“创新”出来之前的某段历史，把对比的参照物拉回到很多年前，来证明自己的“进步”。

“敏捷”的宣传中，一开始会描述“瀑布”如何如何糟糕，然后说“敏捷”如何如何比“瀑布”好。把“敏捷”直接和“瀑布”对比，似乎在“敏捷”出现之前就是“瀑布”。

实际上，增量迭代开发、持续集成等软件工程实践早已有之。即使是那篇被频繁引用的著名“瀑布”论文，Winston W. Royce 发表于 1970 年的“Managing the Development of Large Software Systems”，也并不提倡以顺序模型作为团队的过程模型。图 1-40 是该文的一个截图，注意图中的“Iterative (迭代)”一词。对全文感兴趣的读者可自行搜索阅读。

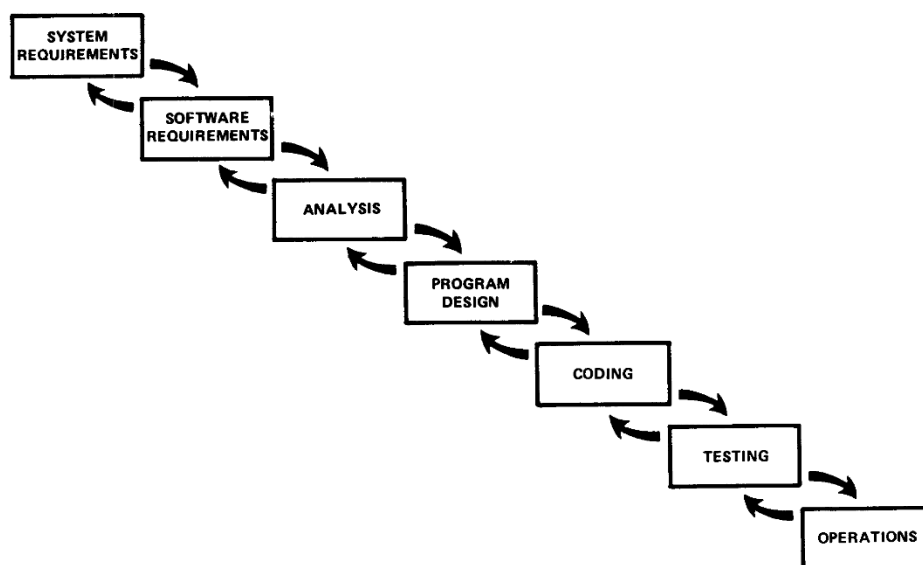


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

图 1-40 摘自 Managing the Development of Large Software Systems (Royce W. W. , 1970)

这样的宣传，抹杀了造词“敏捷”之前的软件开发过程所取得的进展，给不了解情况的开发人员留下“这是敏捷发明的，所以嘴上喊着敏捷的人最懂”的印象。

“领域驱动设计”的宣传中，会说“面向过程”如何糟糕，“贫血模型”的假面向对象如何不好，或者说以前的方法是如何只关心“技术”不关心“业务”，似乎“领域驱动设计”之前就是一片空白。

Eric Evans 可以说是始作俑者，他在“Domain-Driven Design”的前言中说：

Leading software designers have recognized domain modeling and design as critical topics for at least 20 years, yet surprisingly little has been written about what needs to be done or how to do it.

领先的软件设计人员认识到领域建模和设计的关键性已经有至少 20 年，然而令人惊讶的是，关于需要做到什么或者如何做，**一直以来几乎没人写点什么。**

对 Eric Evans 以上言语的批评，可参见我写的文章《DDD 浮夸，Eric Evans 开了个坏头》(umlchina.com/url/evans1.html) 以及《领域驱动设计割裂历史，哪里有详细一些的真实历史？》(umlchina.com/url/dddhistory.html)。

Eric Evans 说的 20 年，指《领域驱动设计》出版时间 2003 年前面的 20 年，大约是 1983-2002 年。事实和 Eric Evans 所说的在这个期间“几乎没人写点什么”恰好相反，去翻翻那个年代的各种面向对象

方法学书籍，“领域驱动”味道比今天还要浓。

这是令人感到讽刺的。既然以“领域驱动设计”为名，按道理应该是领域逻辑越复杂的系统越需要“领域驱动设计”，但领域驱动设计“大行其道”的地方却是面向大众的互联网公司，鼓吹者所举例子的领域逻辑也非常简单，从内容上看，鼓吹者也不像是掌握了面向对象建模方法。参见《是不是互联网更适合用DDD》(umlchina.com/url/internetddd.html)。

究其原因，就是讲述“ $X \rightarrow aX$ ，偷梁换柱”手法时说过的：大家热烈讨论“领域驱动设计”，心照不宣地假装所有人对“传统”的面向对象分析设计已经熟悉到都不屑一谈了——一块掩盖自己无知的遮羞布。

最后附上一段来自领域驱动设计圈子的创新观：

当一个观点、方法或物体被某个人或团体认为是“新的”的时候，它就是一项创新。创新的定义和它是否为客观上的新、是否为第一次使用等关系不大；个体对它的反应觉得它是否属于创新，如果人们认为是新颖的，它就是创新。

1.5.4.5 封闭引用

伪创新圈子在“创新”时，不会系统化地检索文献，而是按照自己的接触面或爱好来引用文献。他们优先把引用停留在“圈子”内，例如 Robert C. Martin Series、Martin Fowler Series，对“圈子”外的贡献所知甚少或视而不见。

这样，就形成了一个互吹互捧的封闭圈子。张三造（山寨或染色）了一个新词，他自己不吹，由李四、王五、赵六来帮着吹，李四造了一个新词，他自己不吹，由张三、王五、赵六来帮着吹.....更离谱的是，有的人写文章，所引用的文献，作者全是自己公司的同事。

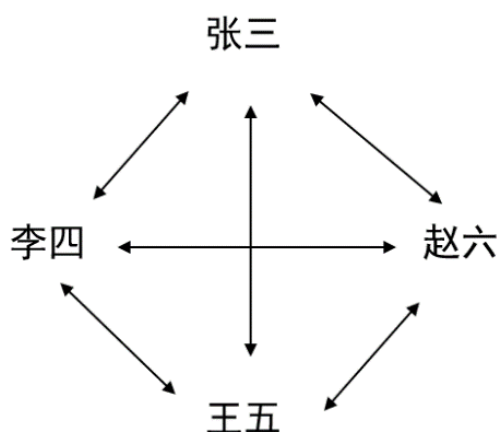


图 1-41 封闭圈子引用

Eric Evans 的《领域驱动设计》，参考文献共 25 篇，图 1-42 是前 17 篇。

参考文献

- [1] Alexander, C., M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams. 1975. *The Oregon Experiment*. Oxford University Press.
- [2] Alexander, C., S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.
- [3] Alur, D., J. Crupi, and D. Malks. 2001. *Core J2EE Patterns*. Sun Microsystems Press.
- [4] Beck, K. 1997. *Smalltalk Best Practice Patterns*. Prentice Hall PTR.
- [5] Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
- [6] Beck, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley.
- [7] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.
- [8] Cockburn, A. 1998. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison-Wesley.
- [9] Evans, E., and M. Fowler. 1997. "Specifications." Proceedings of PLoP 97 Conference.
- [10] Fayad, M., and R. Johnson. 2000. *Domain-Specific Application Frameworks*. Wiley.
- [11] Fowler, M. 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.
- [12] Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- [13] Fowler, M. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- [14] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison-Wesley.
- [15] Kerievsky, J. 2003. "Continuous Learning," in *Extreme Programming Perspectives*, Michele Marchesi et al. Addison-Wesley.
- [16] Kerievsky, J. 2003. Web site: <http://www.industriallogic.com/xp/refactoring>.
- [17] Larman, C. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR.

图 1-42 摘自《领域驱动设计》清华大学出版社 2005 中译本

[1][2] 的作者 Alexander 不是软件人士, [3][7][10] 的作者不算在“敏捷”圈子里。除此之外, 全在“敏捷”圈子里 (对的, Larman 也是)。

★还留了 3 本不在圈子里的, 已经算不错的了。现在圈子里的文章, 恨不得 100% 封闭引用。感兴趣的读者用“架构”这样一个中性的词搜索, 看看热门文章里面的内容和参考文献, 封闭到了什么地步。

就拿被多次引用的 Martin Fowler 来说，Fowler 没有赶上 1980-1990 年代那一段面向对象分析设计方法学兴起的时期（所以没有 Fowler 方法学），他的第一本书 “Analysis Patterns” 出版的时候已经是 1996 年，当时 Fowler 32 岁。Fowler 在书中也称自己为 “后来者”。

“Analysis Patterns” 中使用的是 James Odell 的方法学，模式可以看作原创的。Fowler 后来写的书涉猎甚广，但原创度不如 “Analysis Patterns”。

引用 Martin Fowler 的书是可以的，但不应该停留在 Martin Fowler，否则就会产生前面所说的 “(20 年) 一直以来几乎没人写点什么” 的错觉。

例如，David C. Hay 有一本和 “Analysis Patterns” 类似的 “Data Model Patterns”，在 “Analysis Patterns” 中也有引用，可惜不在圈子中，很少有人提。

★下面这段花絮属于写到此处时的随想，将来会删掉。

★我的遭遇和 David C. Hay 类似。最近一年我不是在翻译 “Analysis Patterns” 嘛，[《分析模式》漫谈](#)写了 60 多篇文章 (umlchina.com/url/aptalk.html)，把书上所有的模型图全部改编成了 UML，并把一部分重要的模式用代码实现了。这些都是之前的译本没有做到的。你猜怎么着，这一年来，DDD 圈子没有人提起过，当作不存在。

★啥？DDD 圈子凭什么要提《分析模式》？问题是，上一个中译本（2020 中译本），DDD 圈子可是大力宣传的，已经把它纳入领域驱动设计的领地了！

★在此也一并回答有同学提过的问题：《分析模式》漫谈，为啥有时要顺带评点领域驱动设计？因为 “Analysis Patterns” 原书没有提到 “领域驱动设计”（这个词还没出来呢），2004 中译本中也没有提到 “领域驱动设计”，而 2020 中译本中则出现了 “领域驱动设计”。

1.5.4.6 用口号代替方法

以下做事情的方式：有口号有方法、有口号无方法、无口号无方法，哪一个方式最坏？

可能有的人会认为无口号无方法最坏，其实不然。无口号无方法地呆在原地，可能会慢慢衰落，但不是最坏的。

历史上各种最坏的事情往往和 “有口号无方法” 有关——最坏的事是 “有口号无方法” 的 “好人” 做的。

“坏人” 知道自己做的是坏事，会暗自收敛，事后会内疚，甚至做一些善事来弥补以求心安。比如，强盗打家劫舍抢了一千万，可能会拿出五十万来 “济贫”，剩下九百五十万自己爽。这样，他心里就觉得自己是 “侠盗”。

而 “好人” 认为自己是做好事，既然做好事，那出手就不要再有顾虑了，做得越绝越好，所以可能会做得很极端。如果有口号无方法，大悲剧就发生了。

软件开发行业如果不警惕“有口号无方法”，可能会有以下危险：

(1) 无意识的自我陶醉

很多产品经理会喊口号：我们只做最重要的需求，尽快把系统推向市场；

很多架构师会喊口号：设计要分离变和不变，这样可以减少变更的成本。

问题来了：

怎样知道哪个需求最重要？拍脑袋？

怎样知道哪些变哪些不变？抓阄？

可惜，有的人喊完口号之后，就满足了，陶醉了，觉得自己太牛×了，这么厉害的道理都懂，其他都是小意思了，到矣尽矣，蔑以加矣。他们并不认真思考怎样才能真正做到，连自己以前采用的一些虽然不是很好但还过得去的做法也懒得做了。

200*年，有一位网络名人，北京大学计算机科学博士说过一段话：

关于软件工程，我并不看重。我更相信人对代码的控制能力，正常人对于正常复杂程度的代码，控制几十万行代码应该是可以的；如果软件总体上有很好的结构设计，模块之间有稳定、合理的接口，那么，仅仅依靠人的脑袋，加上良好的编程习惯，即使面对大型的软件，应该也能控制。

问题来了：

“很好的结构设计，模块之间有稳定、合理的接口”、“良好的编程习惯”怎么来的？从天上掉下来的？这不正是软件工程研究的学问吗？上面这段话就是正确无用的废话。

如果不具备基本的经济学常识，不要说计算机科学博士，就算是院士，针对软件开发发表的言论都有可能是错得离谱的。

(2) 有意识的胡作非为

这是最坏的情况。

因为口号是正确的，在口号的遮掩下胡作非为就很难阻挡，而且后果也不好追责。

前文描述过的场景：嫌画类图、状态机图不“敏捷”，一上手就直接编码，然后对着编码环境反复折腾；嫌 UML 不“敏捷”，自己手画或用绘图工具做自编草图，花的时间比用建模工具点几下还多……其实就是反智和掩盖无能的遮羞布，但没关系，“毕竟初衷是好的”。

★有一次我评点客户开发人员提交的工件，顺便批评了他们团队中一些打着“敏捷”旗号的粗陋做法。然后，一位女生天真无邪地问我，“老师，希望敏捷一些难道不好吗？”——我还能怎么说！“敏捷”这个词本身就是褒义的。它占领了正能量高地，让人无法反对。

★此处提到“女生”，没有歧视的意思，只是如果让我想个例子，我印象最深的就是这个。

在陈述事实的基础上，才可能有合理的讨论。像下表中这些，都是陈述事实：

面向对象	面向过程
命令式	函数式
深度优先	广度优先
状态图	活动图
民营企业	国有企业

图 1-43 陈述事实的用词

面向过程好还是面向对象好？或者什么情况下面向过程会好一点，什么情况下面面向对象会好一点？民营企业好还是国有企业好？或者什么行业适合民营企业，什么行业适合国有企业？或者什么发展阶段优先发展哪一个？

这些都可以讨论。

“敏捷”就不一样了，它直接上了一个褒义词。这个褒义词的对面是什么？迟缓？笨拙？你还怎么讨论呢？它已经立于不败之地了。

可能有的人会说，“敏捷”的对面是“瀑布”啊！前面的段落“割裂历史”已说过，“瀑布”对面其实是“迭代”、“增量”这样阐述具体做法的词——这也是本来已经在用的词，但“敏捷”把它们抹掉，直接上一个褒义词来降维打击。

在“敏捷”口号之前，那些后来被称为“敏捷”的过程，叫做“轻量 (Lightweight)”过程。

“轻量”虽然也是一个形容词，但还是比较中性的。就像我们说一个公司是小公司，从员工数量、注册资金或营业额上看，事实确实如此，但如果说公司是“敏捷公司”，味道就大不相同。也许是因为“轻量”的煽动性不够，圈子选择了“敏捷 (agile)”。

★近年圈子使用的褒义词还有“精益”、“现代”、“活”等（欢迎读者补充更多资料）。

1.6. 伪创新的乐土：“互联网”



图 1-43 林正英电影海报

有个巫医发明了一种治疗方法。他坦言，我这个方法对付癌症可能不太行，但对付感冒很管用噢！你不信，找个感冒患者来！

巫医让感冒患者躺在一张绘有八卦图案的方桌上，然后绕患者八八六十四圈，每一圈的唱词都不一样（看到没，他也是有一套方法的！），然后对患者说，回去该吃吃该喝喝，五天之内就好了！

果然，患者好了。巫医四处宣传他的革命性划时代感冒疗法，迅速收获大批粉丝。

真相是：对于感冒来说，喝白水、喝肥宅快乐水+姜、喝盐水、吃草药、吃合成药、巫医施法，无论哪一个，感冒患者要走的流程都差不多，因为它是靠自身免疫力来对抗病毒。

★严格来说，这些也不是完全没有“好处”。可以借机放松身材管理，痛饮肥宅快乐水；吃合成药可以缓解一些症状；巫医施法能带来心理安慰。

1.6.1 “互联网系统”

我们这里说的“互联网系统”，更严谨的说法是面向大众、用户量巨大的网络信息系统。

目前（2026 年）很多信息系统所使用的技术栈、部署方式以及访问方式和“互联网系统”相似，例如某家制药企业内部使用的制药行业生产管理系统，但并非面向大众，用户量也不大，像这样的系统，不是我们这里说的“互联网系统”。

互联网浪潮到来之前，信息系统的竞争焦点是功能。

★我 1997 年硕士毕业，先到高校当了一年老师，然后才去软件公司做程序员。第一个参与开发的系统是酒店管理系统（实现：Visual Basic+SQL Server）。这样的系统用的人不多，服务器一台，每个部门放上一台客户端电脑就差不多了，但功能很多，入住、退房、收银、客房，餐饮、娱乐、财务、电话计费、各种报表，覆盖了酒店内部流程的方方面面。

互联网的兴起带来了这样的系统：系统功能很简单，开发系统时需

要思考的核心域逻辑很少，但是通过互联网，系统可以让非常多的人使用。

早期的典型例子是 1996 年出现的 hotmail (被微软收购后演变成今天的 outlook.com)，这是一个基于 web 的电子邮件系统，推出一年多时间就有 1200 万的用户。同一时期类似的系统还有 ICQ (即时通信)、GeoCities (个人主页) 等。

★注意，邮件系统也是有丰富逻辑的，包括标准化的协议 (SMTP、POP3、IMAP、MIME.....)、地址解析和路由、加密等等，但绝大多数这些逻辑已经被前人探索得很清楚，甚至有实际的可用组件提供，并不需要 web 电子邮件系统的开发人员从头思考和建模。

今天，典型的例子是微信、抖音、微博等，当然，它们用户量往往以亿甚至十亿计。

用户量的增加和功能的简单化是互相促成的。随着系统用户数量的增加，系统的功能往往会取各类用户想要的功能的“公约数”，想要添加点什么必须非常谨慎，不会为了小部分人的特别要求就为系统增加新功能。

例如，有微商希望微信提供“美化营销话术”的功能，能根据对方的工作性质、性格等美化要发过去的信息，让其更容易打动对方.....张小龙肯定理都不理。当然，其他开发组织可以为这些特定人群开发相应的微信小程序，那是另外的系统了，用户量也不是微信的用户量。

反过来，功能简单的系统，如果用户量不大，就没有盈利的可能，也就养不起开发团队，不会出现后文所描述的伪创新危害。

你觉得现在的记事本用得不爽，于是自己写一个自己用，你爱怎么开发怎么开发。只不过，你以此为素材分享你的“敏捷开发经验”的时候，造成的影响很小。

如果你的记事本不小心碰到了哪个痛点，用户量大增，但模仿者也很快出现，这时就进入到比拼背景、人脉、资金的环节。拼不过，“先驱”成了“先烈”。拼得过，成了赢家，这时再分享你的“敏捷开发经验”，影响力就不一样了。

这样的“互联网系统”，如果说有什么“技术难度”，肯定不是功能的实现，而是用户量起来之后，如何保持性能不下降。起大作用的还是金钱——有没有足够的资金不断购买基础设施来支撑到打败竞争对手。

这个“技术难度”也不是竞争的关键因素。微信之前有米聊，后来者还有易信、来往……。微信的对手们之所以在竞争中落败，是因为它们做不出微信的功能吗？是因为它们的技术无法应对用户量的增长吗？

1.6.2 “敏捷”的乐土

很多开发人员进入了开发或维护这样的“互联网系统”的公司，我们就叫“互联网公司”吧。这样的公司就成了很多打着“敏捷”旗号的“方法”（实际上也没啥方法，主打一个“干就是了”）的乐土。

经常有人和我说：潘老师，敏捷这一套做工厂管理系统之类的可能不太行，但不得不承认，做互联网很管用噢！

我还看到，互联网公司的开发人员霸占了各个技术大会的讲台，他们四处布道“互联网开发思维”、“敏捷开发思维”。经常有这样的场景：一个互联网公司的开发人员在台上讲他们怎么开发自家系统的，台下的听众一听，这 TM 不就是瞎搞，不就是以前常说的作坊式开发嘛！当然了，演讲者自称“敏捷试错”、“敏捷开发”。

可是，不服不行，人家公司上市了，而且现在开始盈利了！台下有做电力、税务之类软件的同学就开始反思了，人家“瞎搞（敏捷试错）”，这么成功，我们又什么规范，什么建模，一年辛辛苦苦就赚个辛苦钱，不行，要向他们学习，回去马上引进互联网思维和敏捷开发思维，把我们的研发流程互联网化、敏捷化！

有的公司真的这样搞了，结果搞出烂摊子来，后来又来找我，潘老师，我们还是得把建模这些东西重新捡起来。

首先，可以用“幸存者偏差”来解释。很多“瞎搞（敏捷试错）”的公司死掉了，没有“正在其中工作”的开发人员，更不用说在技术大会的讲台上布道了。

其次，即使对幸存者而言，它们在竞争中能够打败其他对手存活下来，并不是自家系统本身的能力比竞争对手系统的能力高出一截，而是其他因素导致的。布道者为了拔高自己的形象，有意无意把自己所在公司的成功和“瞎搞（敏捷试错）”联系起来（这一点，前面在谈到“伪创新”买家时已经说过），引导听众掉入“错误归因”的逻辑陷阱。

“瞎搞”是事实，“成功”也是事实，但不能把并存当作因果，得出结论“因为瞎搞，所以成功”、“只要瞎搞，就能成功”甚至“只有瞎

搞，才能成功”。很可能该公司的背景、人脉以及烧的钱才是成功的原因，至于公司里的开发团队采用什么开发方法，是站着、坐着、躺着还是倒立着开发，无关紧要，但是布道者巧妙地把并存转成了因果。

有一天，张三喝酒喝得半醉去买彩票，结果一开奖，中奖两个亿。大家请张三上台介绍经验，张三如实描述“我那天喝酒喝得半醉去买彩票，就中大奖了”，于是台下听讲的彩民纷纷去喝个半醉然后买彩票，以为这样就能中大奖。可惜，没中奖的李四王五等人没资格上台布道（幸存者偏差），喝醉是事实，中大奖也是事实，但不能因此推导出因为喝醉所以中大奖（错误归因）。

张三之所以能中大奖，背后肯定有原因，只不过这个原因很复杂，属于“上帝算法”，人类目前还算不清楚（否则用来算一下明天双色球多好），所以常会归因到能观察或理解的事情上。“喝醉酒”是一个用于加深读者印象的极端例子，更常见的归因是张三“祖上积德”。

利物浦和曼城激烈搏杀，最终利物浦 2:0 曼城。为什么利物浦能赢球，原因是球员、技术、战术的问题，哪个环节做对了或者搞砸了，球队主教练斯洛特或瓜迪奥拉心里是清楚的，但这对某些球迷来说太复杂了。他们会从一些自己能理解的地方找原因，比如球员球衣的颜色，教练穿什么内裤，球员在更衣室里是站着开会还是坐着开会，球员是不是结对洗澡等等。

★类似互联网布道者这样的错误归因，在互联网之前的软件开发行业就已经有了。上个世纪，很多企业的信息化改进受到很大阻力，反对的声音说：我们企业那么多年一直这么干，不也一直在赚钱嘛。这就是

把并存当成因果。

★“没有做信息化改进”和“赚钱”是并存的，但之所以赚钱，是因为背后的某些背景。了解改革开放初期历史的同学可能知道“双轨制”、“官倒”这些词，以前某些官员的亲戚办公司赚钱实在太容易了。如果这个权力带来的好处能一直很爽地吃下去，搞不搞信息化改进是无所谓的。之所以要改进，就是因为这个“红利”不能继续吃下去了。

可能有人会说“就算功能很简单，但要应付这么多用户，背后技术门槛也不低啊，要是没有很多技术大拿支撑，网站早就崩了”。

我们再拿醉汉张三中奖的故事来说。喝醉酒和中大奖是并存，但不是因果关系，这是前面所说的。我们再看另一个问题，张三中奖两个亿，必须纳 4000 万所得税，否则大奖就没收而且还要坐牢。假设有个有钱人李四，他看到张三先中两亿大奖然后纳税 4000 万，于是李四干脆就预交了 4000 万给税务局，然后再去买彩票。李四中奖两个亿的概率会大大提高吗？

纳税 4000 万是中奖两个亿的结果，不是中奖的原因。互联网公司的很多开发人员属于“纳税型”，是公司的成功带来了他，他却误以为自己带来了公司的成功——“没有我们，双 11 的时候网站就崩了”。

★如果觉得“纳税”不好听，可以说“保镖”。张三中奖两个亿，然后请保镖，保镖说“没有我，你能有这两个亿吗”。

1.6.3 “领域驱动设计”的乐土

随着互联网（以及移动互联网）的不断扩展，工作和生活的各个流

程中和大众接口的环节，大多数都已经改成了信息系统接口，例如购物流程中的下单环节、就医流程中的挂号环节、政务流程中的申请环节.....。

开发人员开始不得不思考一些领域逻辑，但前面已经“敏捷”习惯了，建模能力普遍退步或者从来就没有建立，也不愿意咬紧牙关认真学习。于是，迎合这些开发人员口味的伪创新翩翩登场。

其中的代表就是“领域驱动设计”。在[1.5.4.1 偷梁换柱]中已有描述：把比较难、懒得学的建模方法归为“传统方法学”，把“领域驱动设计”吹捧为“新一代方法学”，这些开发人员在谈论时把焦点放在“新一代”（或革命性、划时代）上，心照不宣地假装自己对“传统方法学”已经熟悉到不屑一谈了。

所以就出现了这样的诡异现象：既然以“领域驱动设计”为名，按道理应该是领域逻辑越复杂的系统越需要“领域驱动设计”，但领域驱动设计“大行其道”的地方却是“互联网系统”，布道者很大比例来自“互联网公司”。

常见的领域驱动设计文章是这样的：

先念经“DDD 是解决业务系统复杂性的方法学”，然后再把几个DDD造词吹一通，这一套复制粘贴的常规操作做下来，就已经占据不少篇幅了。

接下来开始举例了，但例子只有 1-2 个领域类，涉及的领域逻辑也极其简单（也许在 DDD 圈子看来已经够复杂），然后就开始大谈实体、值对象、仓储、上下文、聚合根、六边形架构.....还会给出代码，

代码一层又一层，颇为壮观。一点点领域逻辑，包出这么多饺子，难怪伪创新买家（开发人员）会大喊“受用”！

也有银行、保险、医疗等行业组织中的开发人员分享“领域驱动设计”的心得，你以为这回应该有真东西了吧？

仍然没有。他们象征性地简单罗列一些领域概念，意思是“你看，我这里有领域喔”，然后还是前面那个套路，最后，声称领域驱动设计给他们带来了多少多少的好处。

领域逻辑哪里是简单罗列一下就能表达，不使用类图、状态机图这样的建模技能（或等价的形式）是很难刻画出来的——但没关系，他们会骗自己以及骗别人，只要我开始编码，领域逻辑会自然而然从天上掉下来。退一步说，大不了就不断“敏捷试错”嘛！

伪创新圈子如果能咬咬牙去学习，掌握了真正有用的建模知识，大概率也就不会再去相信之前那些伪创新话术了，甚至会为曾经相信那些东西而羞愧。但是很遗憾，伪创新圈子很少有人会这样做，他们更热衷于组圈子，搞人情世故。

行业组织中的伪创新买家（开发人员）更值得警惕，特别是他们把手伸向汽车、航空、航天、国防等领域时。

更多详细探讨，参见我写的《DDD 领域驱动设计批评》文集：

umlchina.com/url/ddd.html

第2章 业务建模 之 愿景

这城市已摊开她孤独的地图 我怎么能找到你等我的地方

《模范情书》；词：高晓松，曲：高晓松，唱：老狼；1996

前些年，互联网产品经理圈子流行这样的口号“我们只做最重要的需求”、“砍掉 80%的功能，专注于剩下的 20%”。哇，这个道理真棒！那怎么判断哪条需求最重要？砍掉哪 80%？没有愿景的支持，这些将沦为空话，愿景就是需求排序的主要依据。

以一个目标系统为研究对象，其愿景（Vision）的定义是：

从目标组织负责人的视角看，目标系统最应该给目标组织带来的改进。

要理解好这个定义并不容易，我们从头来理清一些概念。

2.1 系统

2.1.1 “系统”的历史

关于“系统”的研究，在计算机诞生之前就已经有了。

Arthur D. Hall 在 1962 所著的“A Methodology for Systems Engineering”一书中提到，贝尔实验室从 1940 年代开始使用“Systems Engineering（系统工程）”一词，见图 2-1。

The Radio Corporation of America (among others) recognized in the 1930's the need for a systems approach in the development of a television broadcasting service. During and since World War II many operations research groups contributed philosophy and techniques. The RAND Corporation, created in 1946 by the USAF, developed a useful philosophy which RAND called "systems analysis" (Ref 4), similar to what we shall call later the first phase of systems engineering.

In a nationwide survey, Schlager (Ref 5) found that "The Bell Telephone Laboratories, Incorporated, was probably the first organization to use the term systems engineering." If this is true, the use of the term with roughly its present meaning began in the early 1940's. The name was new but the functions were not. "Systems thinking" and the "systems approach," terms which have become popular lately, played a vital part in the Bell System from the very beginning.

图 2-1 贝尔实验室和系统工程，摘自 A Methodology for Systems Engineering, Arthur D. Hall, 1962

Arthur D. Hall 的书还出现了 object、attribute、relationship 等词，如图 2-2。这似乎在说面向对象软件开发，但实际上当时软件开发中的“面向对象”还没有诞生。仔细看文中的 atom、star 等词，可以知道此处说的“系统”并非我们现在当作缺省选项的信息系统，而是广义的“系统——对外能够提供某些服务，内部由若干部件组成的东西，从整个宇宙到一家企业、一个软件系统甚至一个细胞，都可以称为“系统”。

3.2 DEFINITION OF SYSTEM

Unfortunately, the word *system* has many colloquial meanings, some of which have no place in a scientific discussion. To exclude such meanings, we state the following definition:

A system is a set of objects with relationships between the objects and between their attributes.

Objects are simply the parts or components of a system, and these parts are unlimited in variety. Systems may consist of atoms, stars, switches, springs, wires, bones, neurons, genes, gases, mathematical variables, equations, laws and processes.

Attributes are properties of objects. For example, in the preceding cases the objects listed have (among others) the following attributes:

stars—temperature, distances from other stars
switches—speed of operation, state
springs—spring tension, displacement
wires—tensile strength, electrical resistance

Relationships tie the system together. In fact, the many kinds of re-

图 2-2 Arthur D. Hall 书中的“面向对象”

★ “面向对象编程 (object-oriented programming)” 的说法由 Alan Kay 在 1967 年首先使用 (http://www.purl.org/stefan_ram/pub/doc_kay_oop_en) 。 Simula-67 是第一门面向对象编程语言。

2.1.2 系统和容器

本书第 1 章在讲到 UML 和 SysML 在建模中的区别时, 谈到了处理信息的系统以及处理物质、能量的系统的区别。

信息系统接收输入的信息, 使用其内部所封装的逻辑计算得到输出的信息。本书研究的是信息系统, 因此把除了信息系统之外的其他系统, 统称非信息系统。非人生命体 (从大猩猩到病毒) 以及机械 (从老式的汽车到算盘) 都归入非信息系统, 不再细究其分类。

根据以上内容，系统初步分类如图 2-3。

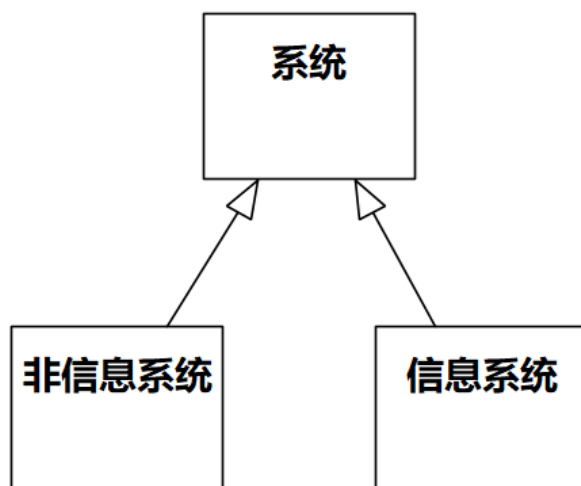


图 2-3 系统的初步分类

★虽然本书到第 8 章才开始讲解类图的建模，但第 2 到 7 章也会用类图来表达概念以及概念之间的关系，目的不是要建模类图，而是帮助理解。如果读者之前没有类图的基本知识，可先阅读本书第 8 章或其他 UML 书籍中的相关章节。其他 UML 书籍推荐《UML 和模式应用》(Larman) 或《UML 精粹》(Fowler)。

根据所驻留的位置，信息系统又可以分为驻留人脑的信息系统（简称人脑系统）以及驻留计算机的信息系统（简称软件系统）。

此处的“计算机”是广义的，是支撑软件系统的基础设施的统称，而不单指某一台计算机的硬件。即使在单机时代，像“浑元太极学员管理系统”这样的软件也已经离计算机硬件有一定距离，更不用说现在的云计算和虚拟化了。

★选择用“软件系统”这个词和“人脑系统”对应，是一个折衷的选择。和“(驻留在) 人脑 (的) 系统”对应的，最直接是“(驻留在) 电脑 (的) 系统”。如果觉得“电脑”港台味浓或者不能用在学术场合，

可以改为“(驻留在) 计算机 (的) 系统”，但“计算机系统”一直以来被认为指计算机本身——类似于《深入理解计算机系统 (CSAPP)》所讲解的内容。如果不使用“人脑系统”这个称呼，可能不会有这些问题，但目前先这样使用，特此说明。

图 2-4 表达了进一步的分类。

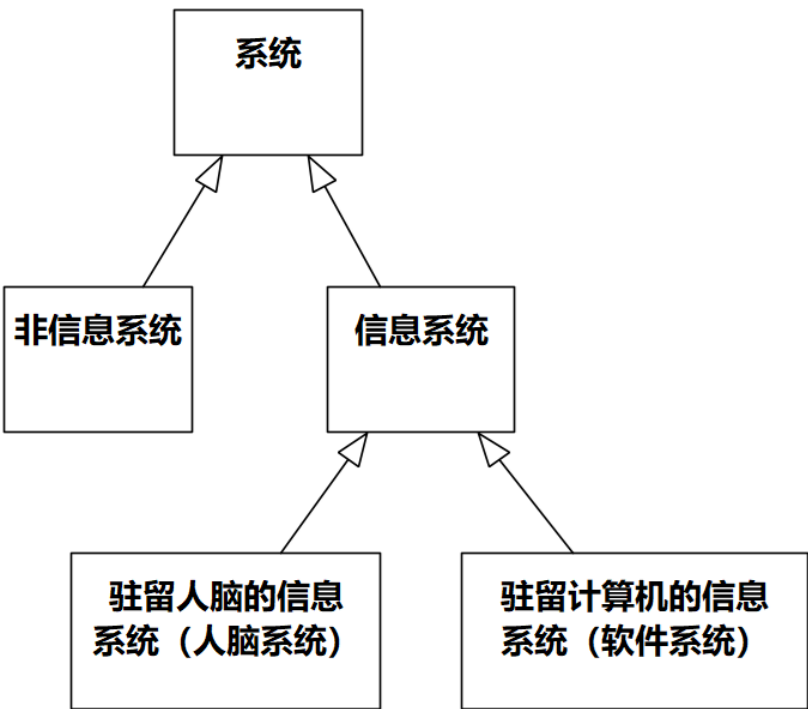


图 2-4 系统根据驻留位置进一步分类

其实，像图 2-4 这样进一步泛化意义并不大，我们更在意的是系统的能力而不是驻留位置。同样能力的信息系统可以驻留在人脑中，也可以驻留在计算机中，人脑中的信息系统可以复刻到计算机，计算机中的信息系统也可以复刻到人脑。哪一个系统能力强，我们就用哪一个。

“人脑”和“计算机”可以看作容器类型，如图 2-5。

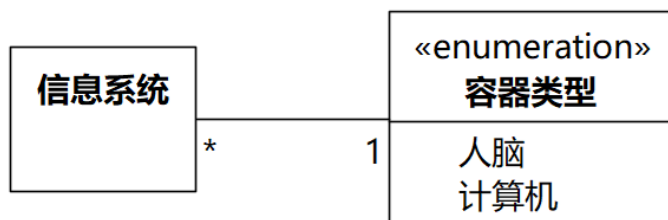


图 2-5 把人脑和计算机看作容器类型

把人脑看作容器之后,人担任的角色或职位就可以看作驻留在人脑中的系统。例如,马宝国是一名.NET 程序员,人称阿宝,同时也是一名浑元太极教练,人称马老师,这可以看作马宝国这个人脑中安装了“阿宝: .NET 程序员”和“马老师: 浑元太极教练”两个系统。

同样,我们把计算机看作容器,上面可以驻留很多软件系统。例如,马宝国所使用的计算机安装了“Visual Studio 2022”和“浑元太极学员管理系统”。

我们把“时间”想象成运行在驻留在上帝中的信息系统,至于它是人脑系统还是软件系统,就要看我们把上帝看作是人还是计算机了。“时间”不停地向其他人脑系统或软件系统发送时间信号,这些系统收到时间信号后,可能会做一些事情。从物理学的角度,这个想象是有问题的,但对于本书的内容来说,这个想象够用了。

这里顺便谈一下面向对象初学者的一个常见错误(包括不久前刚批评的一个领域驱动设计创新“宏论”)。例如,研究一个烹调软件系统,这个系统里面有哪些类呢?有的人会认为系统里有一个“厨师”类,封装了烹调的逻辑。他这样想的理由是,如果没有这样的烹调软件系统,烹调的逻辑是放在厨师的人脑里。

根据上面的剖析，厨师对应的这个人只是一个容器。这个人之所以会烹调，不是因为他是人（我也是人，我就不会烹调），而是因为他的脑子里安装了“厨师”系统，这个“厨师”系统和以计算机为容器的“烹调软件”是对等的。

输入一个信息“宫保鸡丁”，人脑中的“厨师”系统为什么能想到一个烹调的流程，然后按照流程指挥他的手脚以及各种设备烹调？回答“因为系统里有一个厨师类”、“因为他是厨师”虽然很有投资少、见效快、产量高的领域驱动设计风味，却是没有意义的，有意义的回答是“因为系统（即厨师）知道菜品-制作步骤-食材之间的关系”。

同理，输入一个信息“宫保鸡丁”，计算机上的“烹调软件”系统为什么能输出一个烹调的流程，甚至能按照流程指挥电子烹调设备来烹调？回答“因为系统里有一个厨师类（烹调软件类）”也是没有意义的。有意义的回答是“因为系统（即厨师）知道菜品-制作步骤-食材之间的关系”。

实际上，人脑中的“厨师”系统中可以没有“厨师”类。厨师只要记得菜品-制作步骤-食材，忘了自己姓甚名谁并不妨碍烹调。同理，“烹调软件”也可以不需要厨师类，因为它不记住厨师的信息，不影响它输出烹调的流程。

注意：

(1) 是“可以没有”。如果有需要，“厨师”或“烹调软件”当然可以记住厨师以及他和其他几个概念之间的关系。

(2) 注意区分“厨师”和“(供)厨师(使用的)界面”。

第 8 章还会探讨这些问题。

2.1.3 系统和系统规格

由于本书只深入讨论信息系统相关的建模，因此从以下内容开始，我们说到信息系统时只说“系统”，这样可以在称呼上减少一半字数。也许后续讨论的内容其中一部分也适用于非信息系统，但本书就不加以鉴别了。

目前为止，我们对“系统”的定义其实是模糊的。

如果有两名.NET 程序员马宝国和张雪风，他们的电脑都安装了 Visual Studio 2022，这到底算几个系统？如果认为是多个系统，那就出现多个“系统”对象属性值完全相同的情况。例如，很多系统的名称都叫“Visual Studio 2022”，其厂商都是“Microsoft”（如果关注这些属性的话）。

因此，需要区分“系统规格”和“系统”。我们把“Visual Studio 2022”看作一个“系统规格”，某个人脑或计算机容器中安装的“Visual Studio 2022”是一个“系统”。添加上“系统规格”后，类图如 2-6。

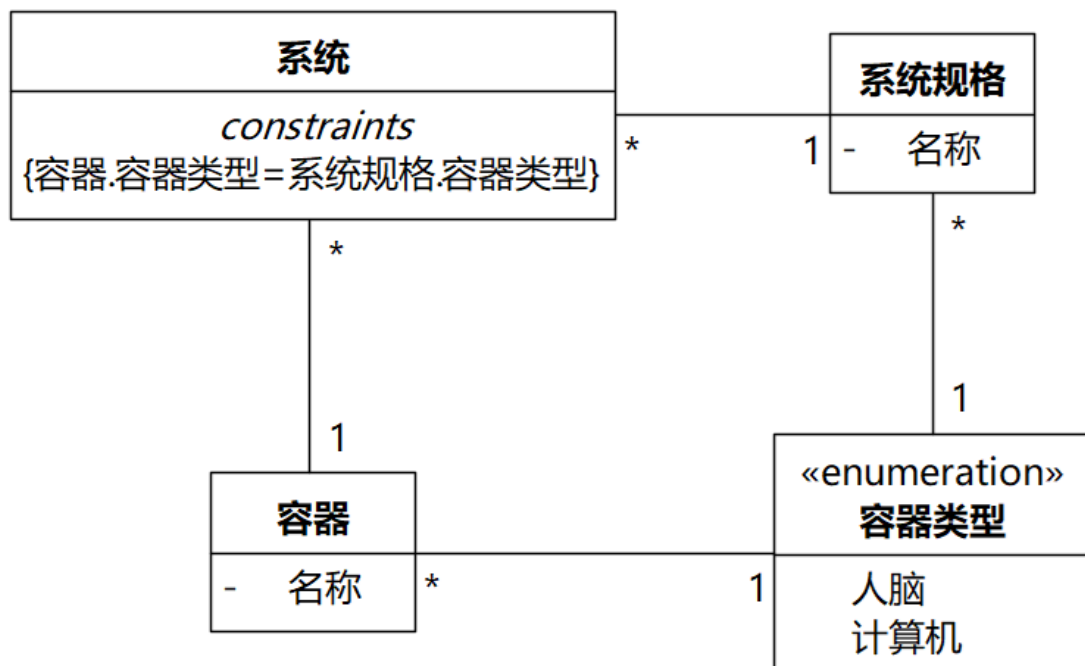


图 2-6 系统和系统规格

用上面的马宝国、张雪风数据实例化图 2-6，得到的对象图如图 2-7。

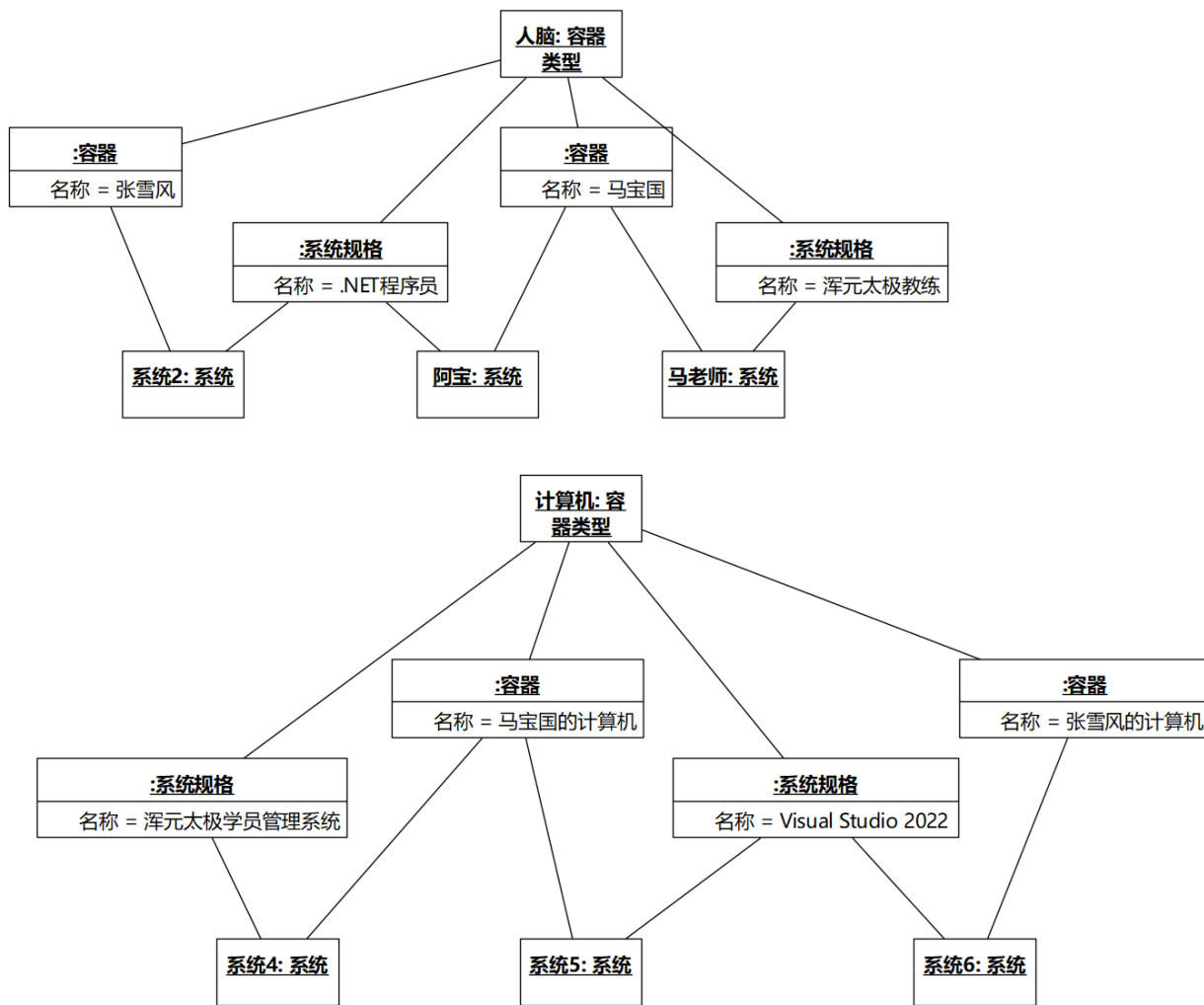


图 2-7 对象图

如果更严谨地追问，上面的模型还是存在模糊。

我们把 “.NET 程序员” 和 “Visual Studio 2022” 看作是对等的，但马宝国后面可能会使用 “Visual Studio 2026”，或者改用另一个厂商出品的.NET 集成开发环境 “Rider”。那么， “.NET 程序员” 最合适的对应应该是 “Visual Studio 2022”、“Visual Studio” 还是 “.NET 集成开发环境” ？

这个模糊的根源可能是：我们比较清楚 “Visual Studio” 和 “Rider” 的差别、“Visual Studio 2022” 和 “Visual Studio 2026” 的差别以及 “Visual Studio 2022 社区版” 和 “Visual Studio 2022 专业版”

的差别，但是不清楚“.NET 程序员”之间的差别。

这个问题我们暂时搁置，以上所提炼的概念对于本书的建模够用了。

后续的建模过程中，我们在研究系统时会有意忽略容器。我们只要知道有两个人脑系统“.NET 程序员”和“浑元太极教练”就够了，它们是不是驻留在同一个人脑中，或者说这两个岗位（角色）是不是指向同一个人，是无所谓的。软件系统也是如此。

所以，在描述系统之间的协作时，我们可能会像图 2-8 这样建模：

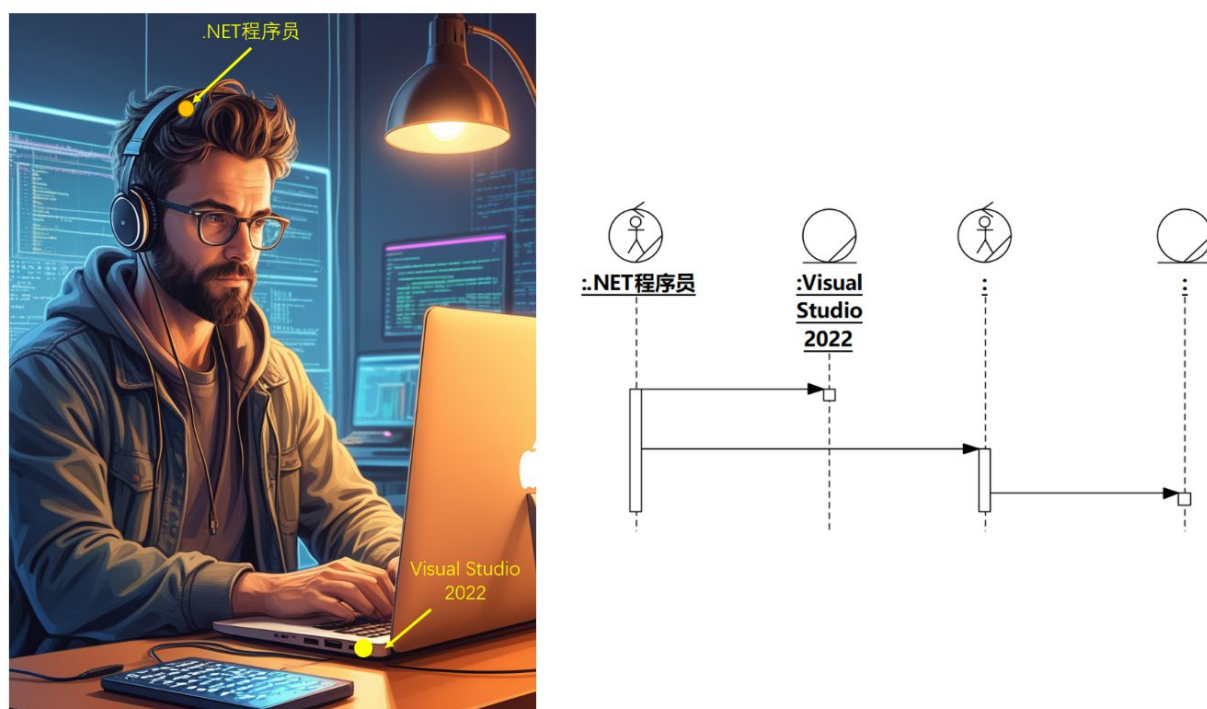


图 2-8 “.NET 程序员”和“Visual Studio 2022”的协作

“.NET 程序员”和“Visual Studio 2022”安装在容器（人脑或计算机）中，这两个系统事实上并不直接接触。如图 2-8 左侧的场景，两者的接触点是人的手指和键盘。人的大脑指挥手指去敲击键盘，键位的变化被捕获并翻译成相应键的编码……但这些都属于容器相关的行为，图 2-8 右侧的序列图不用表达。

容器类型也不会经常被提到，因为从系统规格的名称就可以判断，例如，“.NET 程序员”是人脑系统，“Visual Studio 2022”是软件系统。

2.2 组织

2.2.1 组织和组织规格

和系统一样，我们区分“组织规格”和“组织”。

例如，“县区教育局”是组织规格，“天火市稻积区教育局”是该组织规格的一个组织；“企业人力资源部”是组织规格，“大米公司人力资源部”是该组织规格的一个组织。

一个组织规格由若干系统规格组成，其中至少有一个系统规格是人脑系统（可以只有一个）。也就是说，如果没有人，就算堆满了机器人，都不算我们这里说的组织。

可以给“组织规格”加约束来体现这个规则：

系统规格 $s \rightarrow \text{exists}(s \mid s.\text{容器类型} = \text{容器类型}::\text{人脑})$

用类图表达如图 2-9：

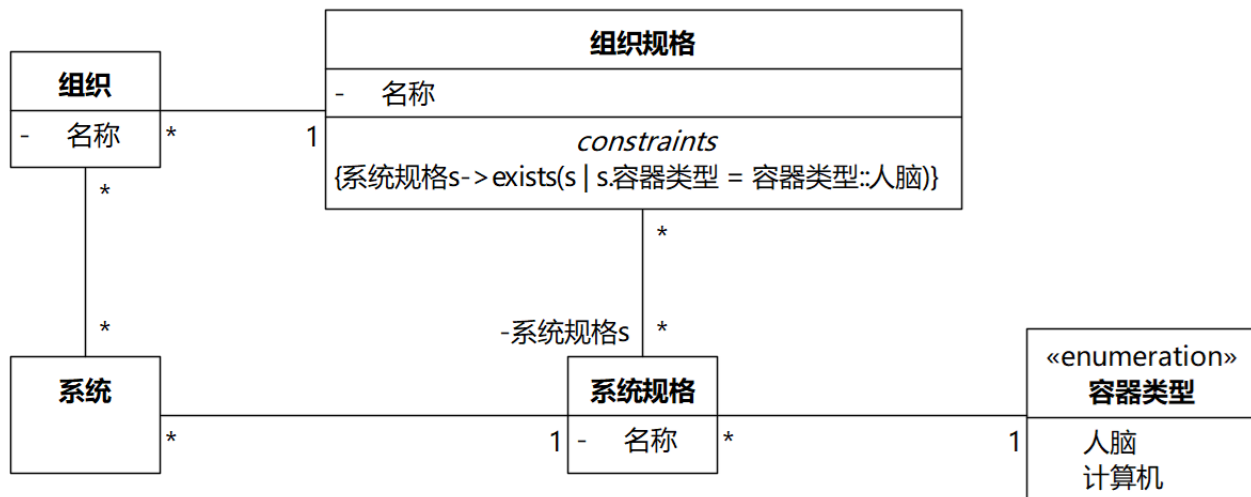


图 2-9 组织和组织规格

从组织价值的视角看，当前组织里面的系统，有的是合适的，有的是应该更换的。更换人脑系统，就是裁员和招聘，更换软件系统，就是.....大家熟悉的场景。组织里面还有非信息系统，也要更换，但这部分本书就不涉及了。

2.2.2 组织规格的分类

可以对组织规格做一些分类，像图 2-10 这样用泛化关系表达：

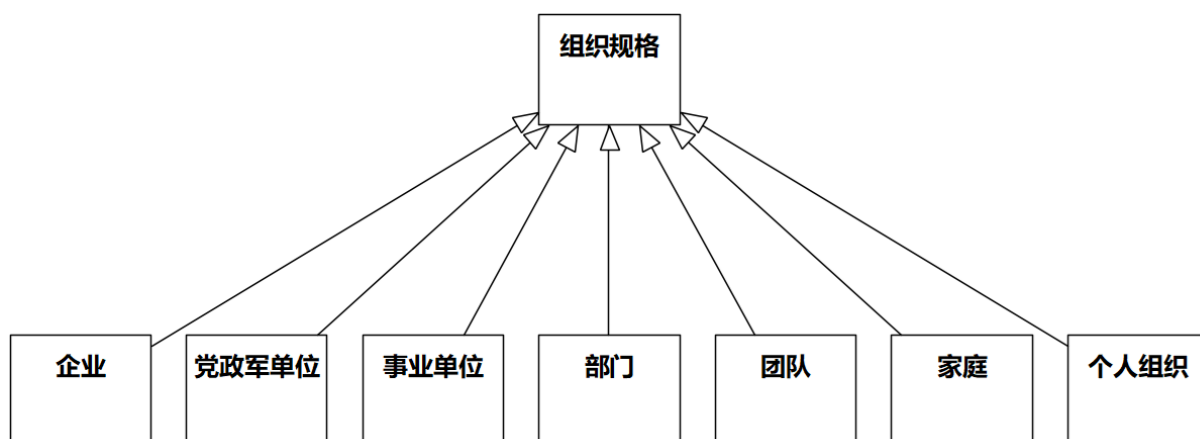


图 2-10 泛化关系表达组织规格分类，空间有限，只表达一部分
在图 2-10 的子类中，有一个“个人组织”。我们把“人以群分”

视角下的个人看作一个组织,该组织的部件包括一个人脑系统和若干非人系统。

例如，“程序员”可以看作一个个人组织规格，该组织规格的部件包括一个“程序员”人脑系统规格，还有“编码工具”、“UML 建模工具”、“版本控制工具”、“AI 工具”等非人系统规格，如图 2-11 所示。

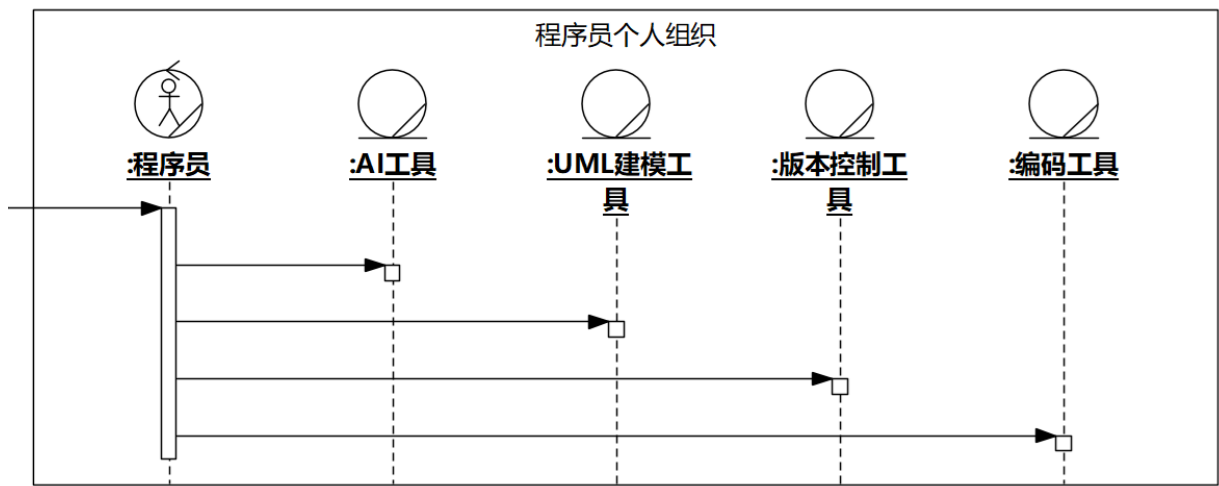


图 2-11 程序员个人组织和程序员人脑系统

这个地方的阐述和以前的《软件方法》版本有了根本的不同。以前的《软件方法》版本把“人群”和“机构”看作对等，可以说是《软件方法》方法学的一个大缺陷。漏了“机构群”这个抽象级别无法对应，同时，生硬地把“人”看作“人群”中的业务工人，也不合理。

这个版本的阐述通过把个人及其所利用的其他工具看作是一个“个人组织”，应该补上了这个漏洞。法律上也有类似的做法，通过“法人”的概念，把自然人和机构拉平，成为对等的民事主体。

注意区分个人组织和只有一个人的机构。有的部门在某个时间点只有一名人类成员，甚至有的企业在某个时间点只有一名人类成员，例如暂时没有另外增加雇员的“一人有限公司”或“个体工商户”，但这样

的组织并非人群视角下的“个人组织”。

图 2-10 所示子类的上面和下面还可以加泛化层次。

往上走，企业、党政军单位、事业单位（以及社会团体、民办非企业单位）等，可以泛化出“正式组织”，它们都有“信用码”属性；部门、团队、家庭、人群视角的个人等可以泛化出“非正式组织”。

往下走，“企业”、“党政军单位”等等，下面还可以分为更细的子类。

泛化的用处是体现行为的变化。本书无意探讨不同子类的组织规格在行为上有什么不同，所以我们把泛化结构取消，用“组织类型”代替，如图 2-12：

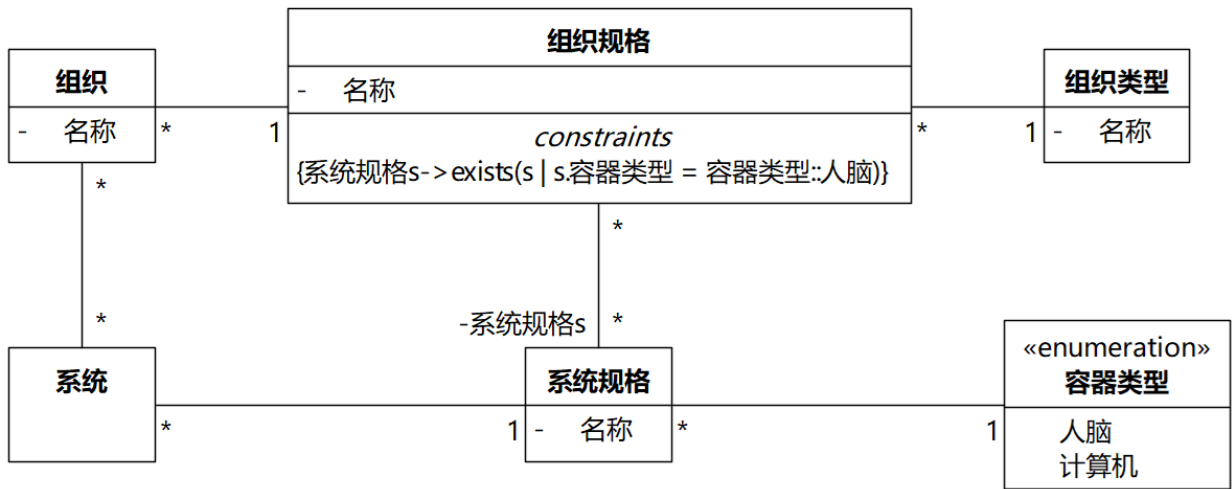


图 2-12 用组织类型代替组织的泛化结构

2.2.3 开发组织和目标组织

在图 2-10 和 2-12 中，“组织”和“系统”之间只有一个关联，所以我们没有给关联或关联端的角色命名。如果要命名，应该如图 2-13

中的某一个：

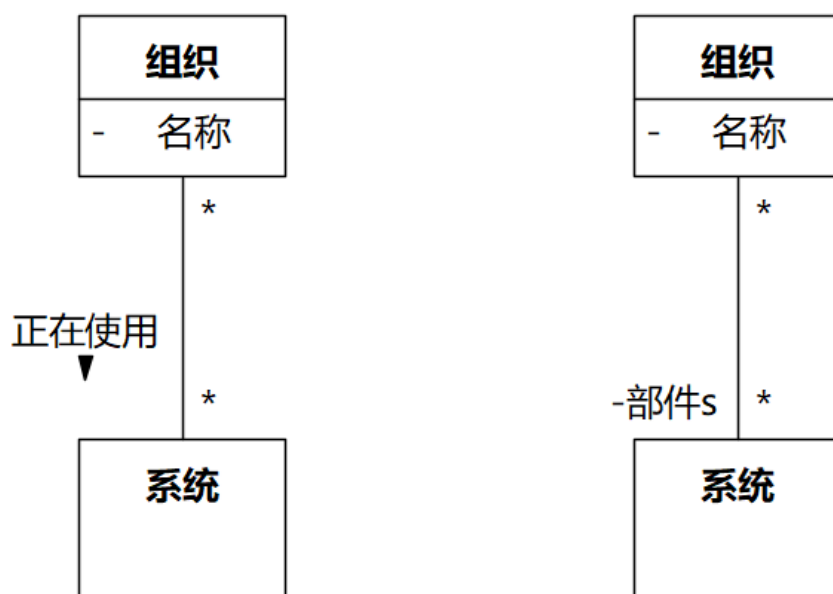


图 2-13 给图 2-10 和 2-12 中“组织”和“系统”的关联添加细节

除了“系统正在担任组织的部件”之外，系统（或系统规格）和组织可能还有其他关系。

山东枣庄有一家“武德物业”，总经理是房竹韧。“武德物业”服务的楼盘有“中原裕府花园”、“拉斐公馆”等。“武德物业”目前使用“银蝶物业管理系统”来协助各项工作。“银蝶物业管理系统”由“银蝶软件”开发。

软件公司“浑元科技”的总经理马宝国看到了商机。他认为，类似“武德物业”的组织目前还有很多可以通过信息化来改进的地方，这些组织中，以“武德物业”最为典型。

于是，他想要做一个“以和为贵物业管理系统”，取代“银蝶物业管理系统”为“武德物业”服务。“浑元科技”有开发团队，但资金不

足，马宝国拉来了“五连鞭资本”加入。

从上述内容看，系统（或系统规格）和组织有以下关系：

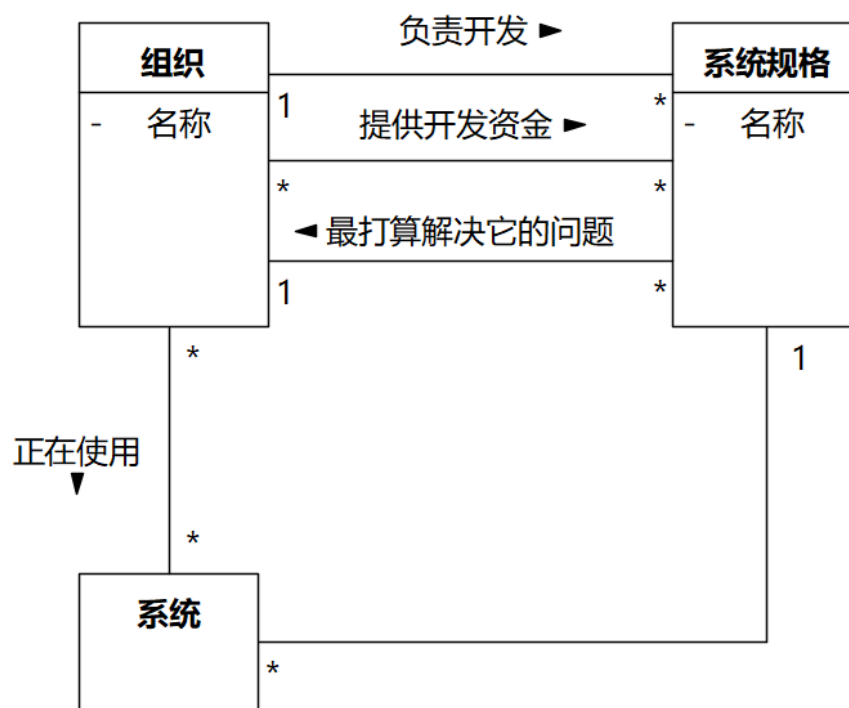


图 2-14 组织和系统的更多关系

用刚才举例的内容实例化图 2-14，得到图 2-15 的对象图：

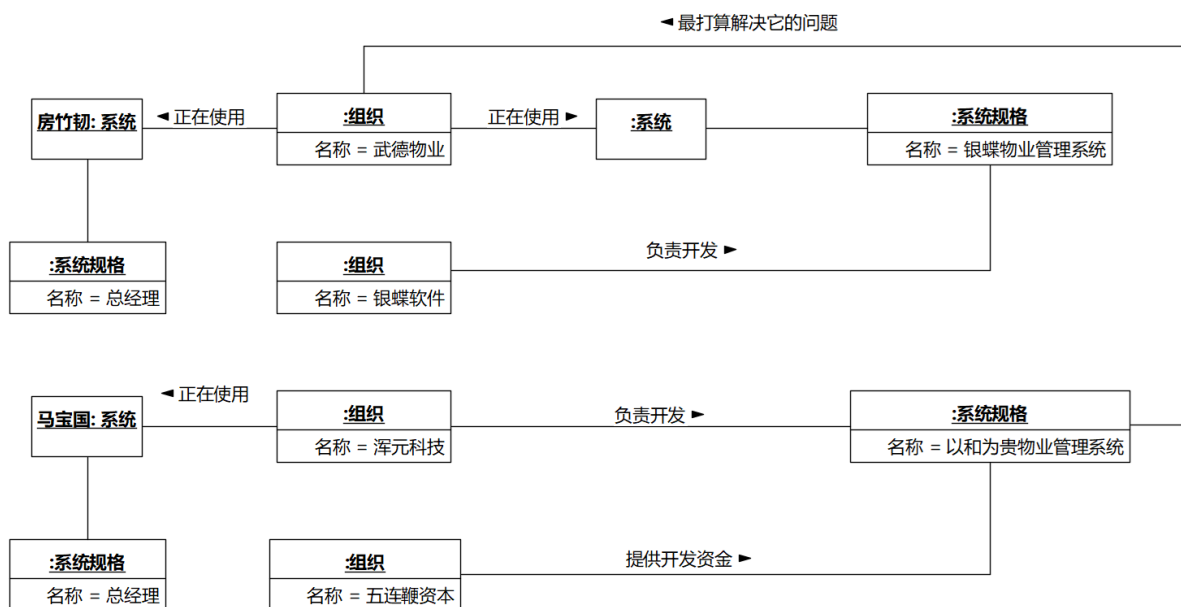


图 2-15 实例化图 2-14 得到的对象图

图 2-15 中，有两个系统规格的实例名称同为“总经理”，这两个不是同一个系统规格。一个对应武德物业总经理的职位，一个对应浑元科技总经理的职位。

有了以上的建模作为铺垫，我们可以定义以下概念：

(1) 开发组织和开发组织负责人

负责构思和生产某个系统规格的组织，称为该系统规格的开发组织；该组织的负责人就是开发组织负责人。

我们不进一步区分出钱、出力、出**资源的组织，如果资源来自多个组织的合力，可以把开发组织视为这多个组织合并而成的一个大组织。上面描述的例子中，可以把“以和为贵物业管理系统”的开发组织看作“浑元科技+五连鞭资本”。

不过，大多数情况下不需要考虑“五连鞭资本”。做事需要钱，钱投进来是为了赚更多钱，这是一个和特定领域无关的普遍道理。如果出资方不是直接指引关于目标系统规格的各种决策，可以当作不存在或者普遍存在，否则很容易变成批量刷废话。

所以，如果马宝国没有被出资方架空，“以和为贵物业管理系统”的开发组织设为“浑元科技”即可，马宝国是“以和为贵物业管理系统”的开发组织负责人。

其他和特定领域无关的普遍资源也是如此考虑。例如，不管开发什么系统，都不能违反法律，否则都没法出现在市场上，谈何盈利？而法

律是由全国人民代表大会制定的，所以，全国人大是所有系统的最重要涉众——但这样的思考对特定系统来说没有意义。

(2) 目标组织和目标组织负责人

最适合用某个系统规格的系统来改进其组织流程的组织，称为该系统规格的目标组织。

上面描述的例子中，“武德物业”就是“以和为贵物业管理系统”的目标组织，总经理房竹韧是“以和为贵物业管理系统”的目标组织负责人。

在《软件方法（上）》的前两版中，目标组织负责人也被称为“老大”。从这一版开始，我们将尽量使用“目标组织负责人”，不使用多余的词汇。

某个系统规格的开发组织和目标组织可以是同一个组织。

组织 A 里有部门 A1（通常是 IT 部门），A1 开发了系统规格 x，用于改进 A1 自身的流程。这时，系统规格 x 的开发组织和目标组织都是 A1。

组织 A 里有部门 A1，A1 开发了系统规格 x，用于改进组织 A 的流程。这时，系统规格 x 的开发组织是 A1，目标组织是 A。A1 属于 A，也可以认为开发组织和目标组织都是 A。

下面这个就不一样了：组织 A 里有部门 A1，A1 开发了系统规格 x，用于改进部门 A2 的流程。这时，系统规格 x 的开发组织是 A1，目

标组织是 A2，开发组织和目标组织是不同的，不能说它们都是组织 A 的部门，所以是同一个。

用医学来类比，看看是否可以帮助理解：

一名心脏内科医生自己检查自己的心脏；

一名心脏内科医生请另一名心脏内科医生帮自己检查心脏；

一名心脏内科医生在医院院长的命令下给全院员工检查心脏；

一名骨科医生觉得自己心脏不舒服，挂了本院一名心脏内科医生的号。

不过，心脏长在谁身上，心脏当前状态好不好，怎样才会对心脏好，这是有医学事实和医学规律的，不以谁给谁看病而变化——这一点后面还会不断强调。

开发组织和目标组织是组织和特定系统规格关联时所扮演的角色，所以在提到开发组织和目标组织时，务必要指明是哪一个系统规格的开发组织和目标组织，除非已经有一致的认识。

平时我们说的甲方、乙方、客户等，是很含糊和简陋的说法，很多错误就是由这些含糊导致的。

很多开发人员和我说，我们这是一个内部项目，没有客户。噢，那你们可以为所欲为了？想想就不可能嘛。所以，在下面的内容中，除了描述某个开发组织的具体场景时可能会使用之外，我们在严谨叙述方法学时，会尽量避免用甲方、乙方、客户这些含糊的词汇。

2.3 建模步骤 A-1 定位目标组织（可选）

在定位系统规格的目标组织和愿景时，有两个思考的出发点：从开发组织负责人出发和从目标组织负责人出发。

在上面所举的“以和为贵物业管理系统”例子中，说“浑元科技”的总经理马宝国看到了商机，这是以开发组织负责人作为思考的出发点。

还有一个可能：“武德物业”的总经理对自己组织的现状不满意，打算引进信息系统来改进，这是以目标组织负责人作为思考的出发点。

平时所说的“做项目”，有一定概率出现第二个情况，而平时所说的“做产品”，都是第一个情况。

第二个情况可以不需要定位目标组织，直接快进到定位系统的愿景。当然，思考的起点必须来自真正的目标组织负责人。“武德物业”IT部门的经理可能会觉得“武德物业”需要引进信息系统来改进“武德物业”各方面的工作，但“武德物业”已经超出了IT部门经理的管辖范围，因此这样的“觉得”是不算数的。

定位目标组织和目标组织负责人（老大）的步骤如图 2-16，我们缺省认为使用了 AI 来辅助，后文在描述接下来的建模步骤时也是如此。

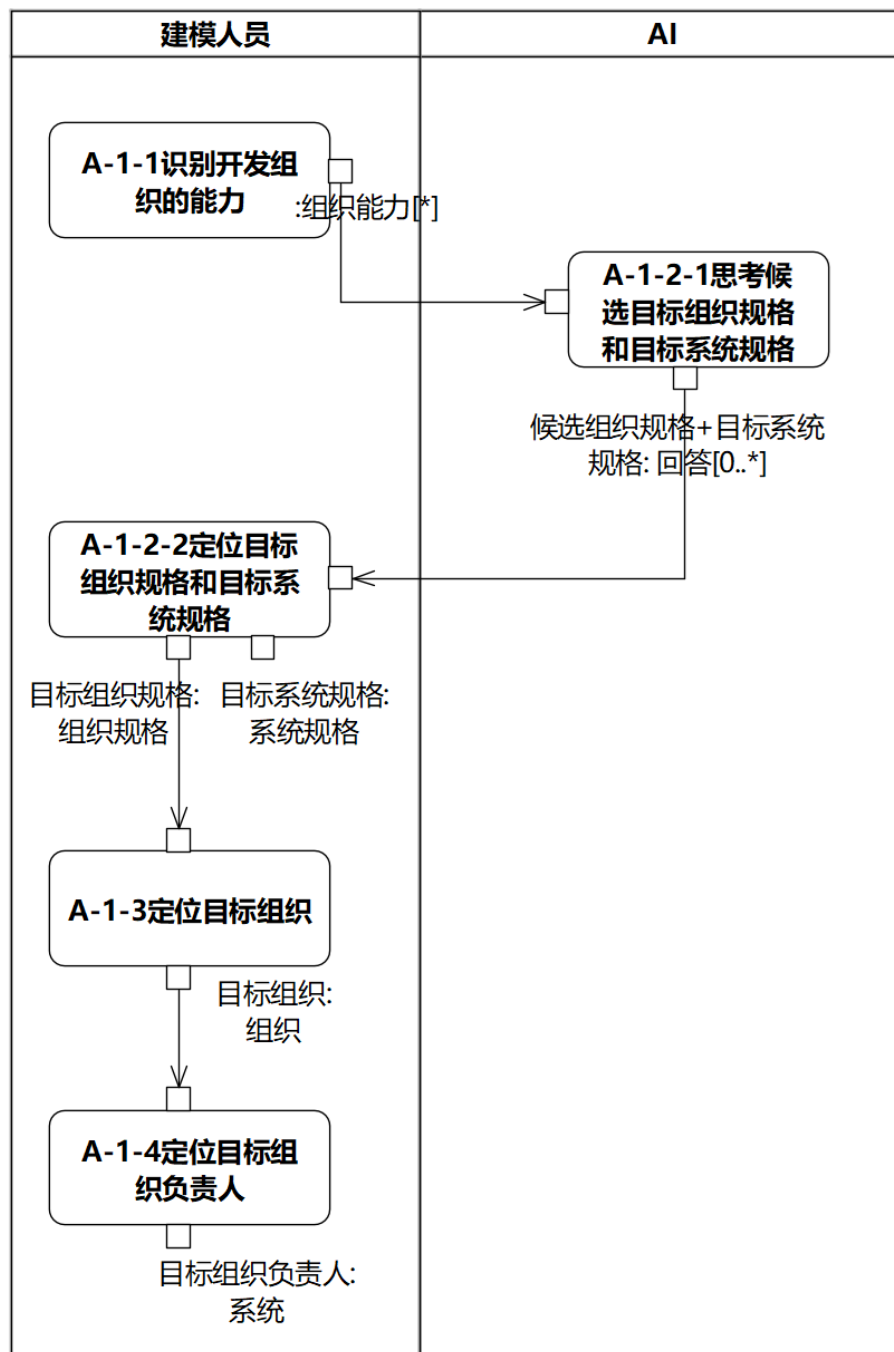


图 2-16 定位目标组织和目标组织负责人的步骤。注意，目标组织负责人的类型是“系统”。

★在之前的《软件方法》版本中，目标组织负责人被称为“老大”。

2.3.1 步骤 A-1-1 识别开发组织的能力

开发组织负责人就像军队的指挥官，他决定军队当前最应该开往哪

个战场，和那里的敌人搏杀。A 城富庶，打下来可得巨额钱粮，但敌人重兵布防，拿下来的概率很小；B 城敌人力量薄弱，拿下来的概率很大，但 B 城很穷。

要做出最佳的决策，首先要开发组织负责人对自己组织的能力有清醒的认识。

下面是描述开发组织能力的一个例子：

接发软件公司自评有以下能力：

**Java 技术能力，能力等级为领袖等级。开发团队核心成员是 Apache 顶级项目的代码贡献者。公司有一个内部研发的框架“左正蹬（ZZD）”，水平处于业界前列。*

**农业信息化能力，能力等级为卓越等级。做过十几个农业信息化的项目，在病虫害预警、精准灌溉和产量预测方面积累了丰富的经验，但业界还有体量更大、背景更深的竞争对手——耗子尾汁科技。*

**烟草业人脉，能力等级为优秀等级。公司总经理马宝民的伯伯最近调任省烟草专卖局局长。*

可以看到，能力不仅包括软件开发能力、行业知识能力，还包括人脉背景等。

可以用类图表达组织能力相关概念如图 2-17：

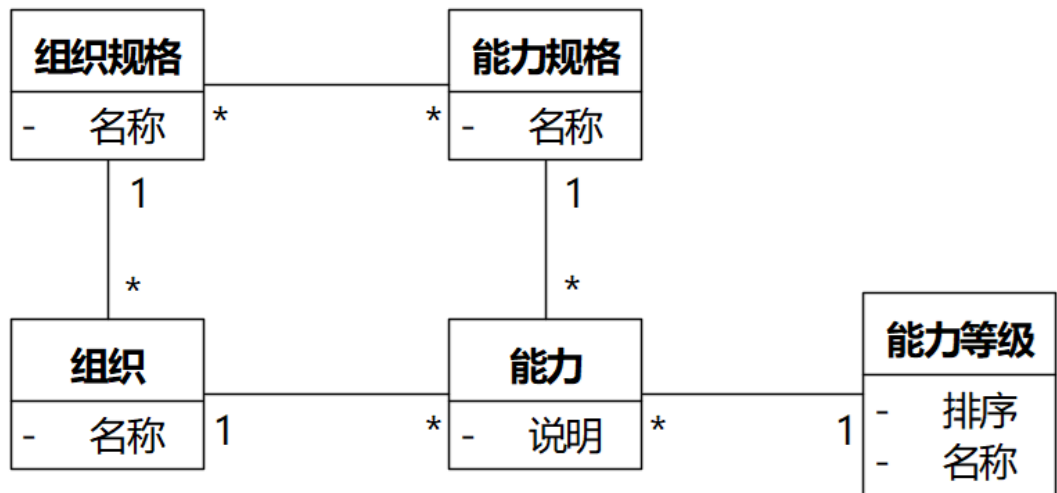


图 2-17 组织能力类图

注意，图 2-17 的类是“组织”，不是“开发组织”。组织有这些能力就有这些能力，即使它躺平不扮演开发组织，也有这些能力。

用接化发软件公司的数据实例化图 2-17，得到对象图如图 2-18：

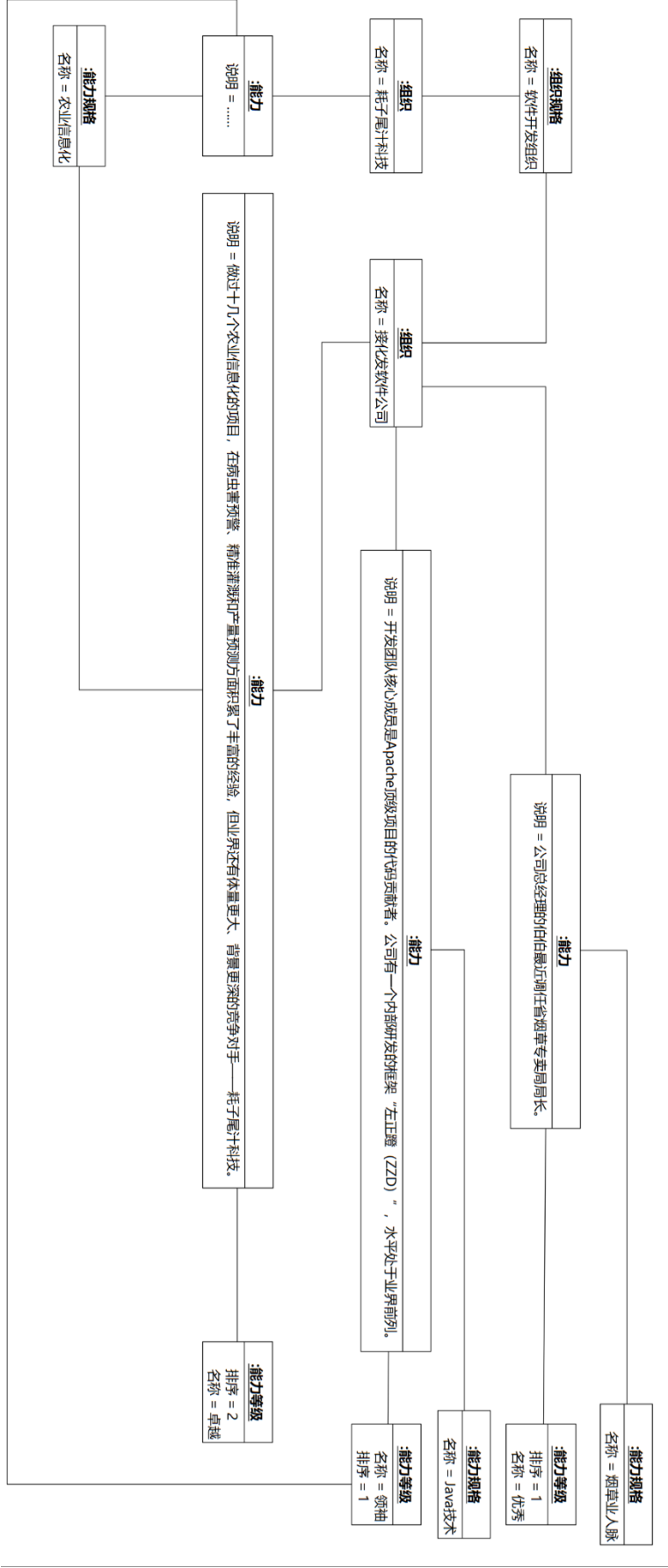


图 2-18 组织能力对象图

2.3.2 步骤 A-1-2 定位目标组织规格

2.3.2.1 放下幻觉，尊重现实

识别开发组织能力之后，接下来就是从能力来定位目标组织规格。如果开发组织打算做一个系统来改进某个组织的流程，最合适的组织应该是什么样子？

根据我接触各种各样开发组织时的所见所闻，大多数开发人员，甚至有一部分挂着“产品经理”头衔的开发人员，头脑中根本没有定位的概念。

在给某团队剖析他们当前正在做的项目时，我问负责的产品经理，谁最适合用这个系统？

如果是目标系统为某个组织定制的系统，也就是所谓“做项目”，这个有现成答案的，产品经理还能回答得上来。

如果是非定制的系统，即所谓“做产品”，回答可能就是这样的：

对于有明显行业性质的产品，例如“医院信息系统”，产品经理回答：医院最适合用。

对于没有明显行业性质的产品，例如“通讯软件”，产品经理回答：谁都可以用。

这两个回答都是“正确无用的废话”，其中没有任何的思考。如果挂着“产品经理”头衔的人这样回答，而且丝毫没有觉得有什么不妥，问题就比较大了，很可能会造成大量的资源浪费。

如果这一点上不做任何思考的根源是偷懒摆烂,危害还不是最大的。

危害最大的是这个**自大心态**:干嘛要想这个?我做产品,当然希望用的人越多越好了!

这是把幻觉当成了现实。

幻觉:所有组织、所有人都用他做的系统。这个幻觉不需要任何思考,傻子都知道这是好事来着。

现实:目前还没有人用他做的系统。这就需要考虑怎样做才能挖到第一桶金,这个思考可不容易!

如果顾客的额头上贴着一份招标书,上面写着资质、评分标准等信息,产品经理会清醒过来,原来这里也是有门槛、有竞争的、有花销的。遗憾的是,顾客只是把这份招标书默默地藏在心里。

除了自大心态之外,也有**自卑心态**。

有的开发组织负责人以为,有资格和有必要考虑定位问题的开发组织,应该是有较强实力,有雄心壮志的组织。这些组织可能想做产品赚大钱,或者想精准地去投标大项目。自己的组织目前处于苟活状态,靠人脉接点小项目维持,有资格和有必要想这个吗?

只要以开发组织负责人作为思考出发点,这个思考都是有必要而且有资格的。

靠人脉接点小项目,也不是说谁的项目都接。为什么接了二舅舅的,不接大姨妈的?可能大姨妈的项目接不到,因为竞争者和大姨妈那边的关系更近;可能大姨妈的项目对自己这边来说很难,怕做得很烂大姨妈

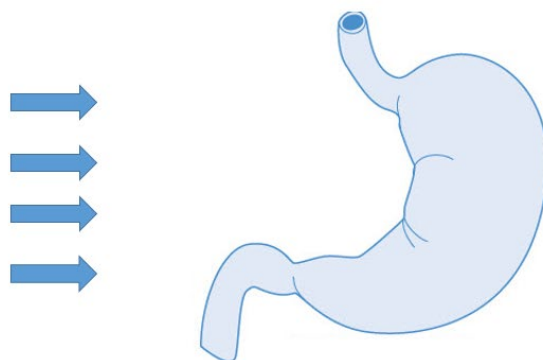
那边都说不过去；可能大姨妈那边太抠门；可能觉察到大姨妈的项目有陷阱.....

就算是乞丐讨饭，也不是对所有的施舍都会感激涕零。

2.3.2.2 形成定位意识

在原始人眼里，喝水是很简单的事情，也没多少选择，靠着河就喝河水，靠着泉就喝泉水。随着社会的发展，“喝水”变得越来越复杂。

“我进超市去买几瓶水带在路上喝”，您猜会买什么？图 2-19 展示了超市里摆的各种“水”（严格来说应该叫：主要成分为 H_2O 的可饮用物质），它们瞄准的目标顾客并不完全相同。



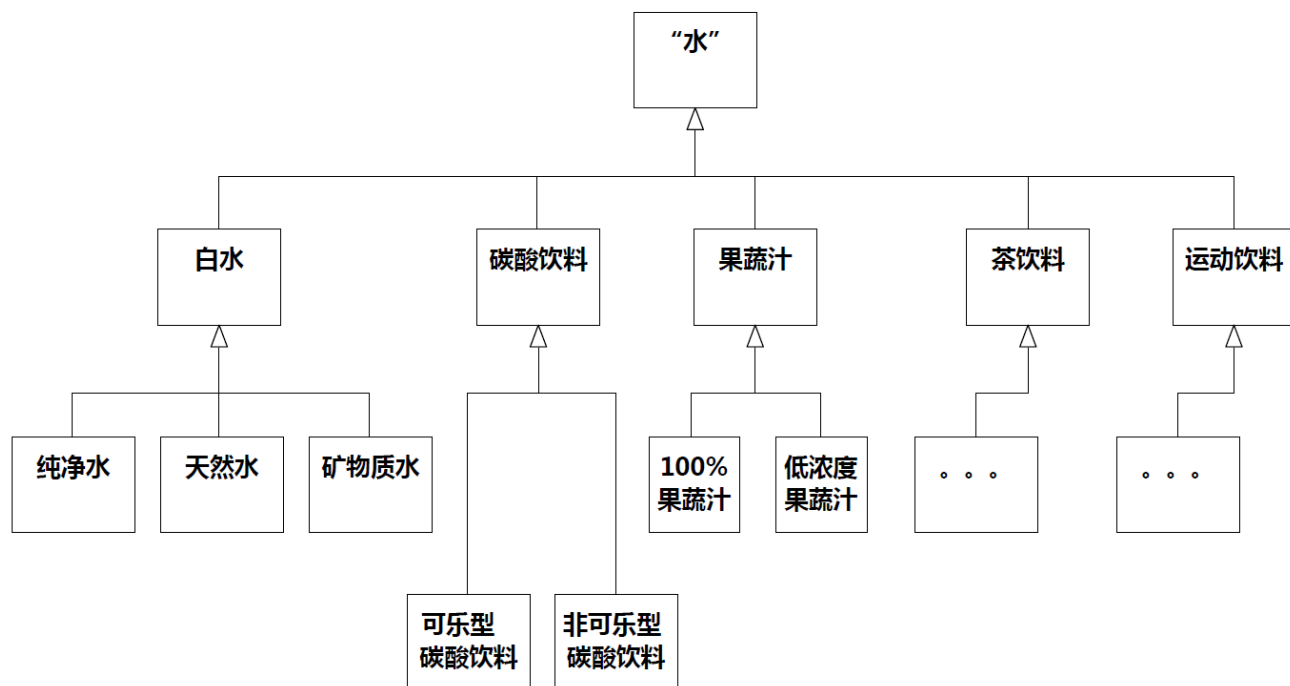


图 2-19 超市里摆的各种“水”抢夺顾客的胃

62 岁的秦大爷找到 KK 可乐公司，和公司负责人说“我喝你们家 KK 可乐几十年了，特别是夏天喝冰可乐，气泡在舌尖上破裂的感觉，那叫一个爽！不过现在我年纪大了，有胃病，还有糖尿病，所以我希望你们家产品在保持原有优点的同时，又不会加重胃病和糖尿病。你看啊，我这里有一个方案，用食品纳米技术把益生菌做成纳米胶囊，这些胶囊在口腔的温度和唾液作用下破裂，又有类似碳酸气带来的爽感，又不伤害肠胃……”

KK 可乐公司负责人会不会很感动，哇，这么棒的老顾客，把要求提得那么具体，实现方案也给了，省下好多我们调研需求和研发实现的时间，好，下个版本就这么办！

我觉得 KK 可乐公司负责人大概率会先感谢秦大爷的精彩建议，然后置之不理。此时的秦大爷已经不属于 KK 可乐的目标顾客（用近几年的时髦词语说就是“基本盘”）。只要年轻的肥宅足够多，他们喝可乐时

依然感到快乐，KK 可乐公司就不会改。

如果秦大爷坚持冒着胃病和糖尿病加重的风险继续购买可乐享用，可乐公司依然会卖给他，只是不理睬他提的建议而已。

如果有一家 A 公司投入资金，按照秦大爷的思路研发一款“纳米可乐”，并以较高的价格卖给秦大爷这样的人（可以把这个人群称为“退休精致大爷”），那也是可以的。这时候这款“纳米可乐”和原来的 KK 可乐已经不是同一产品，面向的目标顾客也不同。A 公司为什么乐意做？有可能是在争取原来的目标顾客时竞争不过可乐公司，只好换一个战场。



图 2-20 不同的“可乐”面对不同的目标顾客

当然，KK 可乐公司也可以考虑做这个“纳米可乐”留住老顾客，但要考虑可能的风险：做出讨好“退休精致大爷”的举动之后，“年轻肥宅”会不会对品牌心生反感。

甚至不用品牌方主动讨好，非目标顾客自己送上来可能也是灾难。试想一下，一个目标顾客设定为“年轻精英男性”的品牌，突然被老年暴走团的大爷大妈追捧，受到因“女拳”出名的杨女士、王女士追捧，会有什么后果？近两年这样的例子有：某购物网站、某酒、某汽车……。

IT 产品同样需要定位。下面几个看起来功能差不多的产品，小红书的目标人群是热衷于分享生活方式的都市精致女性，Bilibili 的目标人群是爱好二次元文化的 Z 世代，抖音的目标人群是从事非思考性事务工作的年轻上班族。

（以上只是作者的看法，可能是有偏差的。自家产品的目标人群是什么样的，厂商的负责人最有发言权。）

并不是说，秦大爷是男性，不能上小红书，秦大爷是老年人，不能上 Bilibili，韦东奕是高校数学教师，不能上抖音。对于厂商来说，自然是来者不拒，多多益善，只不过不同的人所提的意见，被厂商采纳的概率和程度是不一样的，就像上面所举的秦大爷例子。

定位相当于对需求的源头做了排序，最终目的是推导出目标系统当前最重要的需求。如图 2-21，A 和 B 是两个不同的涉众类型，从他们的利益衍生的系统需求分别是 $A \rightarrow x、y、z$ ， $B \rightarrow z、m、-x$ 。

涉众类型	涉众重要性排序	系统需求	涉众认为的需求重要性排序
A	1	x	1
		y	2
		z	3
B	2	z	1
		m	2
		-x	3

图 2-21 不同涉众类型衍生的系统需求

可以看到，需求中有相同的内容，例如需求 z，不过，在不同涉众看来，需求 z 的排序不同；需求中有不同但可共存的内容，例如 y 和 m；需求中还有不能共存的内容，例如 x 和 -x。

如果有了定位的思考，就能确定需求 x 应该先实现。如果没有定位的思考，那就会出问题：

产品经理为图方便吃窝边草

产品经理刚好和 B 比较熟，认为反正 B “也” 可以用，而且调研 B 更方便。注意这个 “也” 字，相当于秦大爷 “也” 也可以上小红书。

于是，产品经理就按 B 的来，需求 z 应该先实现。宝贵的时间、金钱和人力资源会被拿去优先实现需求 z，而在战场上发挥更大作用的需求 x，却迟迟实现不了，甚至因为在实现需求 z 时资源耗尽而且在战场上节节败退，再也没有实现的机会。

这个 “吃窝边草” 的问题，后面还会继续提到。

这还不是最糟糕的，至少产品经理还是从涉众的角度来决定需求的

排序，只不过涉众找错了。下面这个更糟糕：

被程序员绑架

我们再看图 2-22，加上了“程序员乐意实现程度”的干扰因素：

涉众类型	涉众重要性排序	系统需求	涉众认为的需求重要性排序	程序员乐意实现程度排序
A	1	x	1	/
		y	2	3
		z	3	4
B	2	z	1	4
		m	2	2
		-x	3	1

图 2-22 加上“程序员乐意实现”的干扰因素

程序员最乐意实现的是需求-x，如果被这个因素绑架，需求 x 只能沉入海底了。

至于为什么程序员最乐意实现-x，可能是因为-x 刚好在他的舒适区内，不用费脑子思考，也可能-x 的实现刚好需要用到程序员最近感兴趣的某个“新技术”。为了学习“新技术”和美化自己的简历，程序员对-x 充满热情。

二十年前，一位技术总监的话让我印象深刻：

“知道这两个(和愿景相关度最大的)功能实现难度太大做不下去，在我看来这个项目已经没有价值，但是开发人员还乐在其中，觉得还有其他功能可以做。”

由于开发团队中各方关注的利益不同，需求被迫对实现退让——或者说得严重一点——被实现绑架，这样的场景并不鲜见，毕竟领导不能所有事情都亲力亲为，最后还需要有人搬砖。就像电视剧《康熙王朝》里姚启圣所说：



图 2-23 电视剧《康熙王朝》截图

幸运的是，AI 的降临横扫搬砖职位，大大削弱了实现绑架需求的力量。前些年被嘲笑的“我有一个绝妙的创意，就差一个程序员了”，将来可能就没那么可笑了，因为连“差一个程序员”都没有了。

2.3.2.3 步骤 A-1-2-1 思考候选目标组织规格和目标系统规格 (AI 辅助)

再看图 2-22，凭什么涉众类型 A 排在 B 前面？根本的依据要追溯到开发组织的能力。

在“2.3.1 步骤 A-1-1 识别开发组织的能力”，我们定义了接化发

软件公司的能力。现在，接化发软件公司总经理马宝民需要思考，公司做的软件最应该为什么样的组织服务，最应该改善目标组织哪方面的工作？还有，这个软件应该叫什么？

马宝民可以根据之前所归纳的能力以及对当前竞争态势的了解来思考。现在，AI 有了很大的进步，他也可以借助 AI 的力量，获得一些启发。

马宝民可以这样写提示词：

#背景

我想利用我当前所在的软件开发组织的能力优势开发一个软件系统，并希望通过这个软件系统来获利。利益不一定是金钱，也可以是权力、名声。

我把这个想要开发的系统称为目标系统。目标系统是什么样的，有哪些功能，销售给什么样的人和组织，目前我没有确定的概念。我想请你和我一起，一步步推导出目标系统的这些信息。

#任务

请你根据我所提供的开发组织的能力集合，结合你所了解的市场竞争态势，定位目标系统的目标组织规格以及目标系统可以给目标组织的哪些流程和哪些度量指标带来改进。

目标系统的目标组织规格的定义为：如果把目标系统向这样的组织推销，打败其他竞争对手被这样的组织接受的可能性最大，而且在竞争

中获胜后会有一定的利益。

目标组织规格可以是机构（例如省级税务局），也可以是人群（例如福建茶农）。

#所输入信息的规范

提供给你的信息包含以下内容：

*某开发组织拥有的能力集合，每一个能力给出当前在业界中能力等级。等级从高到低依次为：领袖>卓越>优秀>普通。

#要求输出信息的规范

要求输出的信息应包含以下内容：

*系统名称

*目标组织规格

*所改进的组织流程

*所改进的组织指标

*为什么得到这个结果（如果我指定了知识库或者附件作为参考资料，应重点回答使用了参考资料中的哪些知识点来思考）

按照以上格式，返回至少 3 个结果。

#例子：

附件“定位目标组织规格示例.txt”提供了一个示例，帮助你理解。

#真实输入信息

开发组织：接化发软件公司

能力集合：

*农业信息化能力，能力等级为卓越等级。做过十几个农业信息化的项目，在病虫害预警、精准灌溉和产量预测方面积累了丰富的经验，但业界还有体量更大、背景更深的竞争对手——耗子尾汁科技。

*Java 技术能力，能力等级为卓越等级。开发团队核心成员是 Apache 著名项目的代码贡献者。公司有一个内部研发的框架“左正蹬（ZZD）”，水平处于业界前列。

*烟草业人脉，，能力等级为优秀等级。公司总经理马宝民的伯伯最近调任省烟草专卖局局长。

请你根据真实的输入信息，按照以上要求思考并给出结果。

读者替换“#真实输入信息”段落的内容，就可以把以上提示词用于自己的项目。

Claude-4.5-Sonnet（通过 poe 使用）针对以上提示词返回 3 个候选。第一个候选“烟叶种植智能管理系统”见图 2-24：

接化发软件公司目标系统定位分析

根据贵公司的能力集合和市场态势，我为您分析出以下目标系统方向：

结果1：烟草智慧农业一体化平台

系统名称：烟叶种植智能管理系统

目标组织规格：省烟草专卖局下属各市县烟草公司及其管理的烟农合作社

所改进的组织流程：

- 烟叶种植过程中的病虫害监测与防治流程
- 烤烟质量预测与分级流程
- 烟叶收购前的质量评估流程

所改进的组织指标：

- 烟叶病虫害损失率降低30%
- 优质烟叶（上等烟）比例提升20%
- 烟叶收购定级准确率提高至95%以上

为什么得到这个结果：

这个方向综合利用了公司三项核心能力的协同效应。首先，马总的伯伯在省烟草专卖局的关键人脉（优秀等级）为系统进入烟草行业提供了最直接的渠道优势，这是其他竞争对手难以复制的资源。其次，公司卓越等级的农业信息化能力在病虫害预警和产量预测方面的积累可以直接迁移到烟草种植领域，烟草作为经济作物对精准农业技术需求更高、支付能力更强。最后，卓越等级的Java技术能力和ZZD框架可以保证系统的稳定性和扩展性，满足烟草行业对系统可靠性的严格要求。烟草行业的特殊性在于其垄断性质和充足预算，一旦通过人脉关系切入，竞争对手难以进入，可形成事实上的区域垄断。

图 2-24 Claude-4.5-Sonnet 的回答

考虑到篇幅问题，就不完整复制回答的所有内容了，另外两个方案是：基于 ZZD 框架的农业信息化快速开发平台，面向的目标组织规格为中小型农业软件开发公司和农业科技企业的技术团队；经济作物智能种植顾问系统，面向的目标组织规格为种植高价值经济作物的专业合作社、家庭农场和规模化种植企业。

使用 UMLChina 研发的 AI 智能建模工具“发糕”，可以轻松生成提示词并提交给 AI，然后解析 AI 的 json 格式返回结果，放入软件模型中，如图 2-25：

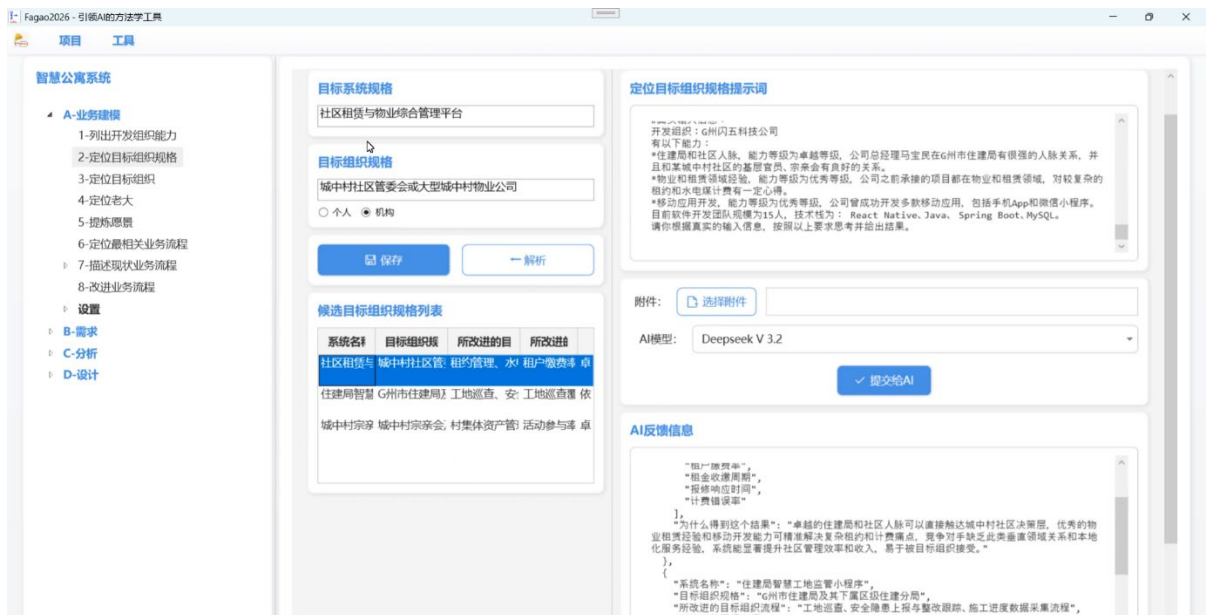


图 2-25 通过“发糕 2026”定位目标组织规格

关于“发糕”的更详细信息，可以到这个链接观看视频：
umlchina.com/url/fagao01.html。

2.3.2.4 步骤 A-1-2-2 定位目标组织规格

AI 返回的候选结果只能作为决策的辅助，如果能够给我们“眼睛一亮”的启发，那自然是最好。如果感觉平平无奇，甚至感觉 AI 在胡说八道，也是正常的结果，毕竟这相当于在问：我这样一个人，做什么赚钱最快？要让答案带来价值，比做数学题和编代码难度大多了。

当然，可以多试几个不同品牌的 AI，提高“眼睛一亮”的概率。

图 2-24 中的“智慧烟叶种植管理系统”（第 1 候选）和图 2-25 中的“智慧烟农助手”（第 4 候选），都是把烟草业人脉和农业信息化能力综合考虑得到的结果，有经验的建模人员应该也会这样来定位。或者说，这样的结果是符合建模人员的预期的。

全部符合预期也未必是好事，说明 AI 没有带来启发。建模人员也

要提防先入为主：AI 给出的结果，凡是和我预期的一致，就是好的，否则就是坏的。如果是这样，AI 就白用了。

既要防止先入为主，又要不被 AI 的胡说八道迷惑，这个尺度如何把握，还得读者自己捉摸。

2.3.3 步骤 A-1-3 定位目标组织和目标组织负责人

2.3.3.1 要真实不要虚构-摧毁偷懒的庇护所

目标组织规格只是定义了规格，符合规格的组织可能有很多个。我们需要进一步回答：符合规格的组织中，哪一个组织最适合用目标系统来改进？在这一步中，我们从很多个组织定位到一个具体、真实的组织。

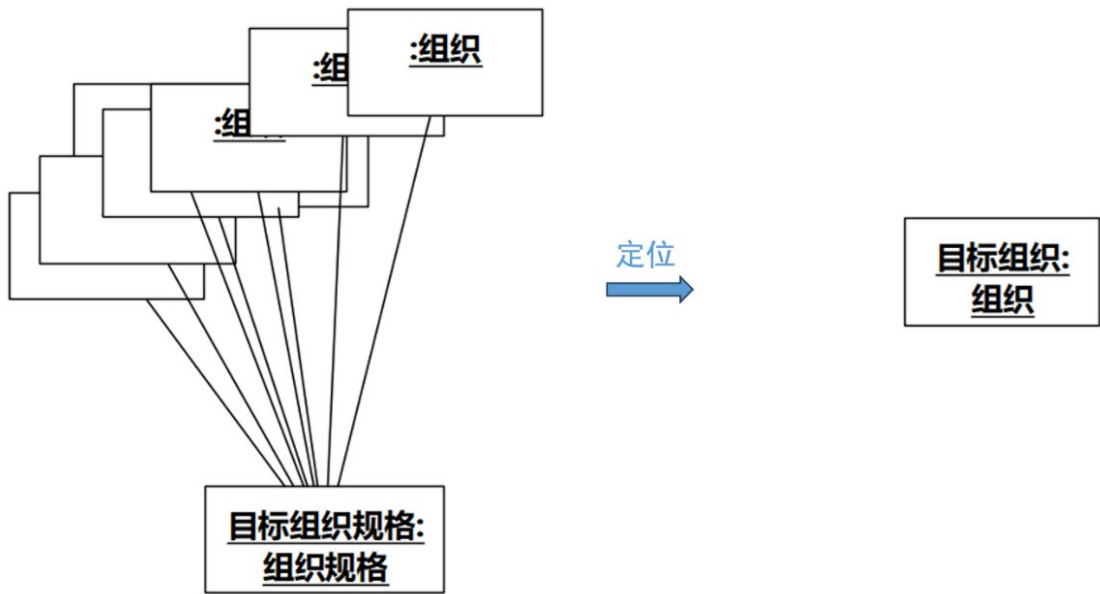


图 2-26 从很多个组织定位到一个组织

例如，图 2-24 提到的“烟叶种植合作社”，全国或全世界的“烟叶种植合作社”数量可不少。哪一家烟叶种植合作社最适合用“智慧烟叶种植管理系统”来改进？

目标组织不但要具体，而且要真实，这有别于虚构的 Persona 或用户画像。

坐在办公室里摆弄各种属性，虚构出一个目标组织（前文已说过，个人被视为个人组织），看起来似乎比一个具体的真实组织更精准——就像概念中的直线比真实世界的直线要直，但这很容易沦为偷懒的庇护所。

定位到一个真实的目标组织，接下来建模人员要辛苦地深入第一线调研。调研的过程是实实在在的，得到的各种文字、录音、录像素材中，可能有一部分素材和建模人员预想的或想要的并不一致，但素材就摆在那里，是不好随便更改的。

如果虚构一个目标组织，建模人员就找到了理由来逃避深入第一线调研的辛苦，因为现实中不存在这样一个“完美”的虚构组织。于是，建模人员安心地闭门造车，想象这个虚构组织的流程，想象这个虚构组织负责人想要的改进指标——更妙的是，他可以随意修改各种数据，直到符合自己的预想。

实在不得不到第一线调研时，建模人员会按自己的喜好挑选一个真实组织来调研，反正大家都不“完美”，我图自己方便挑一个也没关系嘛。

例如，做一款智能项链，如果定位到一名真实的女性：上海浑元商贸公司市场总监 刘艺菲。建模人员需要花时间去观察、访谈刘艺菲，而且有可能刘艺菲的访谈结果会推翻建模人员之前对智能项链的设想，给建模人员增加了不少工作量。

如果虚构一名女性：上海 A 公司市场总监 安风，建模人员就可以肆意想象安风会怎样说和怎样做，这些想象往往会暗合建模人员之前对智能项链的设想——相当于先想好了答案，再不断修改题目来迎合答案。

实在不得不到第一线调研时，由于现实中不存在安风这个人，建模人员可能就会图方便，找隔壁工位的女同事罗玉凤调研，反正罗玉凤也可以用这款智能项链（注意这个“也”字，很多需求问题根源就是它）。

坚持要求建模人员尽其所能定位到具体的真实组织，这是逼迫建模人员深入第一线调研的一个好手段。和其他工作流以及工作流步骤一样，任何逼迫懒惰者思考的做法都会受到阻力，不一定推行得下去，这很正常，但一定要警惕伪创新在其中浑水摸鱼。

★汽车跑得比马车快，但奈何百万马车夫衣食所系，目前不宜推行汽车，这可以理解。伪创新则是把马车改个名叫“有机动力驱动设计全域感知出行终端”，把它包装成革命性的创造、划时代的洞见，宣称汽车过时了。

定位目标组织，可以使用以下思考方法：

2.3.3.2 比较法

列出适用于当前目标组织规格的定位条件，观察和思考哪一个定位条件对于定位目标组织是最重要的。

此处可以用 UML 的类图。以上面提到的“烟叶种植专业合作社”为例，画出类图如图 2-27：

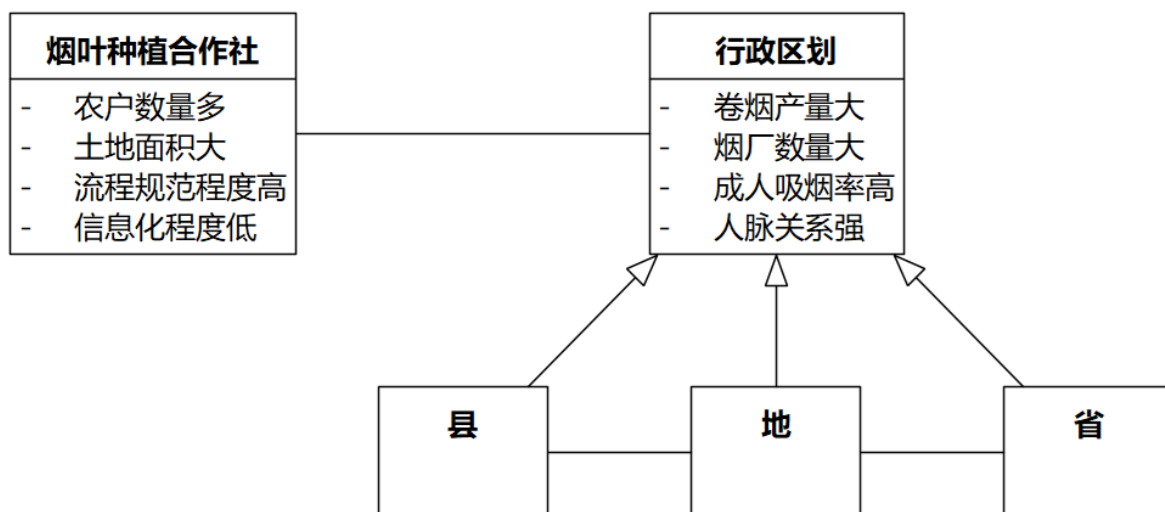


图 2-27 烟叶种植合作社的定位条件

先找对于定位目标组织最重要的定位条件。假设建模人员经过思考，认为“人脉关系强”是最重要的定位条件，从而把范围缩小到最符合“人脉关系强”的 HN 省，接下来，认为“流程规范程度高”是第二重要的定位条件，筛选出最符合“流程规范程度高”的“HN 省 HH 市 ZJ** 烟叶合作社”。

可以看到，定位条件包括一个可以量化的属性（例如：农户数量）以及数值的偏好（大、小、适中……），通过比较找出最符合定位条件的组织。

从零开始来标出这些定位条件，难度很大。例如，看着“烟叶种植合作社”几个字，能想到什么属性？可能会先想到名称、地址、负责人、成立时间、农户数量、土地面积，并不容易总结出“(开发组织和所在行政区划)的人脉关系”、“流程规范程度”这样的定位条件。

建模工具可以考虑复制其他项目为相同名称或近似名称的组织规格设置的定位条件，在此基础上编辑。

可以在图 2-17 基础上添加类，得到图 2-28：

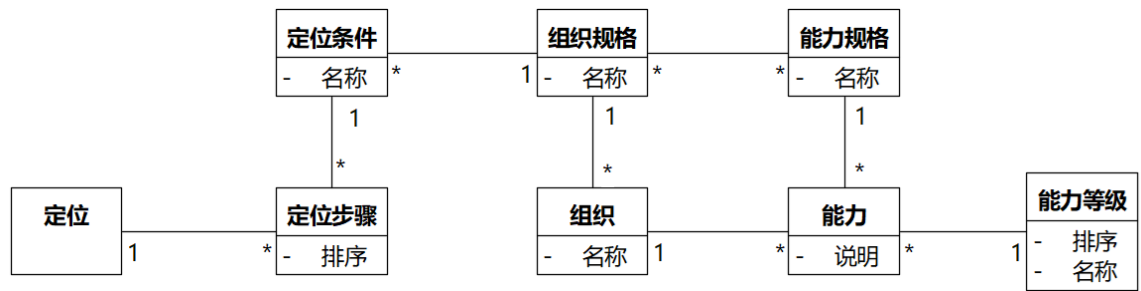


图 2-28 给组织规格加上定位条件

图 2-28 只是记录“名称”属性，没有进一步分解其中的概念。例如，“流程规范程度高”就是“名称”属性的值。

如果要分，还可以从“流程规范程度”抽象出“定位属性”的概念，从“高”抽象出“定位偏好”的概念，这样“定位属性”和“定位偏好”可以随意组合得到不同的条件。有的系统瞄准流程规范程度高的组织，有的系统则反其道而行之，瞄准流程规范程度低的组织，这都是有可能的。

不过，贯彻本书方法学的工具“发糕”在系统中维护定位条件的作用仅用于辅助思考，不打算承担更多的责任——例如从外购的大数据筛选出目标组织，这些交给更擅长的系统去完成。

注意定位条件和能力的区别。例如，“血压高”不是能力，但可能是常用的定位条件。

2.3.3.3 爆炸法

通过直觉先得到初步结果，再通过不断质疑来修正结果。

假设你是建模人员，投资人在你身上绑了炸弹，命令你推销目标系

统并且盈利，而且条件很苛刻：（1）几分钟之内推销出去；（2）只能找一个人推销。

假设这个炸弹能感应推销对象的脑电波。推销完毕后，如果炸弹没有感应到推销对象认为这个系统能解决他的问题并且愿意付出相应代价获得系统，炸弹就会爆炸——既然系统不能盈利，就拿建模人员的命来填。

这时，为了最大可能地保住自己的性命，你会选择向谁推销？这个人就是目标组织负责人，也就是前面两版所说的“老大”。

很多人可能会第一时间想到向自己的父亲或母亲推销，但是，父亲或母亲买单只是为了救你的命，并非认为这个系统能解决他（她）的问题，炸弹依然会爆炸！

如果上面的场景还不足以刺激你思考，可以用加强版：

如果投资人在你和你情敌身上绑了炸弹，命令你们分别按刚才的条件推销目标系统，谁的推销对象对系统感兴趣的脑电波更强，谁就活下来，活下来的人的情敌被炸死。

通过爆炸法初步得到的目标组织负责人，更多是靠直觉得到的，这个结果只是思考的起点，建模人员还需要进一步追问：是什么原因让我选择了他？

以上面的“智慧烟叶种植管理系统”为例，使用“爆炸法”，建模人员凭直觉给出结果“HN省HH市ZJ**烟叶合作社的理事长(社长)”，这时需要进一步追问：

Q: 为什么是 HN 省 HH 市 ZJ**烟叶合作社?

A: HN 抽烟的人多, 烟厂也多。

Q: 就是说, 烟民或烟厂数量多的, 更像目标组织, HN 是最多的吗? 比 YN 怎样?

A:没有 YN 多, 刚才说错了, 是因为我对 HN 这边比较熟悉。

Q: 好, 就是说你是先根据熟悉程度来选择的。HN 这个 ZJ**烟叶合作社是你最熟悉的合作社吗?

A: 只能说相对于其他省份, 我对 HN 最熟悉, 但选择 ZJ**烟叶合作社的原因应该不是熟悉。

Q: 那为什么选 ZJ**烟叶合作社, 它有什么特点?

A: 这个合作社从育苗到烘烤, 流程都比较规范, 而且还没有信息化, 这样的情况比较适合用我们的系统。

Q: 按“流程规范而且还没有信息化”的条件来看, HN 省内, HH 市 ZJ**烟叶合作社是最突出的吗?

A: 以我目前掌握的信息看, 是的。

Q: 假如, YN 有比 ZJ**烟叶合作社更符合“流程规范而且还没有信息化”条件的合作社, 如果让你再重新思考“爆炸法”的回答, 你会不会选择 YN 这个?

A:不会, 现在没有实力让他们接受。

.....

上面的对话中，可以是建模人员自问自答，即提问者（“Q:” 部分内容的提供者）和回答者（“A:” 部分内容的提供者）是同一个人；也可以由一名水平较高的建模人员扮演提问者，本书作者在训练和咨询中经常扮演提问者的角色，不过，水平较高的建模人员费用不低。

幸运的是，AI 非常擅长这样的问答，现在已经可以扮演提问者的角色。

我们可以把上面的套路教给 AI，让 AI 扮演提问者，和建模人员展开问答，推导最合适的目标组织。

提示词如下，替换[]里的系统名称，即可用于你自己的项目。

#任务

我现在正在做一款[智慧公寓系统]，当前目标组织规格定位为[城中村房东]，需要进一步定位具体的目标组织。

请你参考附件“爆炸法.md”中的问答套路，扮演提问者的角色（即“Q:” 部分内容的提出者），和我展开问答，推导最合适的目标组织。

#约束和建议做法

1. 在确认你理解了附件参考资料的内容后，向我提问“请你用爆炸法思考，定位一位你认为最应该向其推销的[城中村房东]负责人。”
2. 得到我的回答后，你继续追问，不断质疑我所定位的具体目标组织和目标组织负责人是不是最合适的，我们将做多轮问答。
3. 当我表示“我认为问答可以终止”时，你停止提问，并以清晰的列表形式，总结出我们共同提炼出的目标组织和筛选条件。

把以上提示词提交给 AI, 问答的过程如图 2-29 (为缩短图片长度, 已裁掉 AI 界面控件部分):

Q: 用爆炸法思考, 你认为最应该向谁推销你的“智慧公寓系统”?

广州宏发达公寓 业主 马宝华

Q: 好的。为什么是广州宏发达公寓的马宝华? 他或者他的公寓有什么特别之处让你第一时间想到了他?

宏发达公寓是在广州天河棠下城中村, 其实是私人自建房, 房东管理手段落后。

Q: 明白了。所以你选择他的核心条件是“城中村的私人自建房”和“管理手段落后”。

那么, 在广州, 符合这两个条件的公寓业主有很多, 为什么你首先想到的是“宏发达公寓”的马宝华? 以“管理手段落后”这个痛点来看, 他是你所知道的最突出的一个吗?

是的

Q: 好。那我们再看看“广州”这个选择。

你之所以选择广州, 是因为广州符合“城中村私人自建房多、管理落后”这个特征的房东数量最多、需求最迫切? 还是因为你对广州这个市场比较熟悉?

一个是熟悉, 第二个租客数量要保证。如果经济不行, 打工人离开了, 没有人租房, 房东也没有动力改进, 所以选择广州。

Q: 非常好。这样就提炼出了一个新的、更关键的条件: “有庞大且稳定的租客数量 (例如外来务工人员), 能保证出租率”。因为只有出租率有保障, 房东才有持续的收入和改进管理的动力。

那么, 同时考虑“城中村自建房多”和“租客数量庞大稳定”这两个条件, 广州是唯一的选择, 或者说是最优的选择吗? 比如说, 和深圳相比怎么样? 深圳同样有大量的城中村和外来务工人员。

图 2-29 让 AI 扮演提问者

和上一步“定位目标组织规格”不同，“定位目标组织”这个建模步骤，公用的 AI 不能一步到位告诉我们，它认为目标组织是哪一个。一是因为它没有数据，二是因为法律问题。

问公用的 AI，哪个城中村出租房管理最落后，AI 回答，我认为是某某市某某路某某号（真实存在）；问 AI，那个年轻人最符合“年轻肥宅”的形象？AI 回答，我认为是某某（真实存在）。这麻烦就大了。

当然，购买某个领域的数据，用私有部署的 AI 或数据处理工具来帮助筛选，不存在上面提到的两个问题。

2.3.4 常见定位错误

2.3.4.1 吃窝边草

“吃窝边草”的错误：建模人员不知不觉拔高“地理位置离开发组组织近”这个定位条件的排序。

以下是本书第 2 版中所举的“吃窝边草”错例，今天仍然有参考价值。这些错例都来自真实的项目。

★XTY 酒家

做餐饮系统，把目标组织定位为“XTY 酒家”。XTY 酒家就在公司旁边，开发团队的同事经常去哪里吃饭。

★隔壁老头

做老年陪伴机器人，把目标组织负责人定位为住在自己对门的 Z 老

先生，毕竟大家是熟人，方便调研。其实 Z 老先生的老伴还健在，他不是最需要这个产品的人。

不知不觉“吃窝边草”，背后的心态是轻飘飘的“***也可以用”。如果前面“爆炸法”的方式思考，轻飘飘的“也可以用”大概率会导致炸弹爆炸，把建模人员炸死。在定位的思考中，必须要思考“最”，警惕“也”这个字。

注意，不要把“地理位置离开发组织近”和“人脉关系强”混淆。上面说的 XTY 酒家和 Z 先生并没有额外给予开发组织一些其他组织所不能给的利益，纯粹是建模人员偷懒图方便。

2.3.4.2 吃自己的狗粮

上面两个例子还不是最离谱的，毕竟开发组织大概率不会兼职开餐馆，开发人员也大概率不是老头，没法说“我也可以用”。或者说，定位定不到自己头上。目标组织就算近在咫尺，也不是自己，还是需要付出一点辛劳去调研的。

下面这个，连这点辛劳都不用付出，想要什么素材，自己编造就可以了：

★我就是幼儿园家长

做一款幼儿园家长用的 app，开发人员的儿子也刚好上幼儿园，干脆就把自己当成目标组织负责人，自己想要什么功能就做，不要的就不做！

软件行业的民粹主义者大力推崇这样的做法，喊出了“吃自己的狗

粮 (dogfooding)”、“工程师文化”等得到广大程序员响应的口号。

★花絮：2025 年 9 月，还有同学传给我一份 DDD 圈子的布道言论：**学习 DDD 最好的办法是一切需求自己说了算。**

有的人还打出 Steve Jobs 和张小龙的旗号来佐证，说 Steve Jobs 和张小龙从来不管客户的想法。其实他们只是不管非目标客户的想法，对目标客户的想法是极端重视的，这正是定位所要达到的目的。

以目标组织的需要为先，可以不管非目标组织的需要。这不能歪曲成谁都不用管，开发团队为所欲为。

实际上，定位目标组织后，开发团队的自由度减少了。之前张三提一个要求，开发团队懒得做，可以找到理由“会损害李四的利益”——总会找到这个李四的。一个事情对大多数人来说利大于弊，这是可能的，但不可能对所有人都利大于弊。如果定下了以张三为先，之前偷懒的理由就不成立了。

本书的前两个版本使用“老大”一词来简称目标组织负责人，有的读者没有仔细阅读“老大”的定义，看着“老大”二字就会产生这样的错觉：最合适的“老大”就是开发组织负责人。他想要做这个系统，才有这个系统，他不想做这个系统，把公司关了改行开饭馆，也由得他。

对于容易被“投资少、见效快、产量高、门槛低、仪式感十足”的伪创新诱惑的开发人员，这样的答案是非常诱人的。它貌似给了一个放之四海皆准的答案，而且不需要任何思考。

是的，想做什么就做什么，不想做也没人拿枪指着你，这无可指责。

该指责的是，你不应该在如此自由飞翔的同时，还期望自然而然地财源滚滚来。懒惰自私是可以的，但懒惰自私的同时又要得多，这就有问题了。

有没有这样的可能，开发组织所定位得到的目标组织规格的代表恰好就是开发组织自身？当然有可能，如果是这样，那就是中了彩票大奖了，是大好事，同时概率也很小。

前面提到的 AI 智能建模工具“发糕”，就是典型的（UMLChina）“也”可以做，（UMLChina）“也”可以用的例子。我们就拿这个来剖析看看：

开发组织：UMLChina；开发组织负责人：我（潘加宇）。

如果我（开发组织负责人）产生了一个想法：做一款封装《软件方法》方法学的智能建模工具，接下来需要思考哪些问题？

首先思考，自己组织的能力做这款智能建模工具是不是最有竞争力？

可能有的同学觉得很荒谬，你自己写的书，自己塑造的方法学，你不是最有竞争力谁是？未必。

能力一：对《软件方法》方法学的理解深度。这个就算我最有竞争力吧。当然，不排除有的同学在读《软件方法》的基础上比我更上一层。

能力二：图形操作、渲染和布局。这个压根没有自己的积累。就算

看不上开源库，花高昂授权费去买更好用的商业库，竞争对手同样可以从第三方免费或付费获得此能力，大家持平。如果和 EA、MagicDraw、Rhapsody 等私有、自研 20+ 年并（很可能）包含一定建模语言语义的图形引擎相比，竞争优势就是负值了。

能力三：手绘识别.....

能力四：利用 AI.....

也许某个成熟的工具厂商再加上一名熟读《软件方法》的建模人员，做出来的“智能建模工具”能轻松打败 UMLChina 开发的工具。

第一步就已经倒下了：UMLChina 做这个未必是合适的。

如果还是坚持做，那么这款智能建模工具可能需要把焦点放在“智能”上，放弃图形相关的部分。幸运的是（以下内容涉及一些暂时不便公开的思考）.....，这一切有了可能。

接下来思考，什么样的目标组织规格，最认为这款智能建模工具能帮到自己，乐意付出代价来获得它，然后开发组织能从中获得满意的回报？这个目标组织规格是以潘老师为代表的“类潘老师人群”吗？

如果“潘老师这样的人”指的是“软件开发方法学家”人群，那就别指望了。

首先，那些不学习不研究的伪创新“方法学家”在忙着造新词和组圈子，怎么可能会踏实思考方法学问题？

“真·方法学家”呢？都方法学家了，肯定要推自己的方法了！他可能会买一个来参考，但更常见的可能是“要”一个。注意，不是南宁周

某“要了几辆电动车”那个“要”，就是人情往来。再说了，就算是买，有几个“真·方法学家”？能收多少钱？

如果“潘老师这样的人”指的是“认同《软件方法》的开发人员”人群，这个人数还算多，但我并不是这个人群中最需要这款智能建模工具的人。实际上，这款智能建模工具封装的知识和我大脑里的知识是重叠的，它起到的作用就是取代我。

也就是说，如果目标组织规格是“认同《软件方法》的开发人员”，目标组织负责人应该是一名《软件方法》水平为中级，有意愿和有权力在团队中推行《软件方法》的开发团队负责人（我大脑里有具体的人，此处隐去），并不是潘加宇。

第二步又倒下了：回旋镖旋回到自己头上，并不容易。

最后，我归纳一下“吃自己的狗粮”的不同场景，大家自行比较。

开发人员张三所在公司的老板让张三开发一款给钓鱼佬用的 app，我们假设盈利模式老板心里有数。张三觉得自己偶尔也会去钓鱼，“也算钓鱼佬”，所以就按自己的想法来。

这是出于自私，图省事而不顾公司利益。

开发人员张三热爱钓鱼，并结交了一群钓鱼佬同好。张三想利用自己的能力做个 app，给兄弟们带来方便，同时自己还能获利。张三认为自己就是钓鱼佬的代表，所以就按自己的想法来。这个场景和上面的智能建模工具类似。

这里面的问题不是自私，为兄弟谋福利倒是可敬的。问题是错估了自己。

自己的能力做这个有没有竞争优势？不好说。自己是不是钓鱼佬的典型？这个很可能不是。和前面所说的幼儿园家长类似，一名职业为软件开发人员的幼儿园家长能代表典型的幼儿园家长形象吗？一名职业为软件开发人员的钓鱼佬能代表典型的钓鱼佬形象吗？

可能有人会辩解说，“吃自己的狗粮”是鼓励开发人员尽量多使用自己开发的系统，感同身受地体察系统给涉众带来的方便和痛苦，并不是让开发人员把自己当目标涉众。

遗憾的是，现实中很容易这样滑坡。开发人员大概率会把自己感受到的方便和痛苦当作目标涉众的方便和痛苦，因为客观上他已经没有更多的精力再去体会目标涉众的方便和痛苦，更要命的是，主观上，他也不愿用目标涉众的方便和痛苦来否定自己感受到的方便和痛苦，而是会说“系统很好用，是你太笨了”……

不可否认，开发人员感受到的方便和痛苦和目标涉众的方便和痛苦有一部分是相同的。比起更烂的情况“开发人员随心所欲开发，对涉众置之不理”，“吃自己的狗粮”肯定是有好处的。

但要在竞争中胜出，思路并不是找“还有没有比我更烂的”，而是要找“还有没有比我更不烂的”。这样的思考难度更大，更耗费脑力，很多人不是愿意去做的。

这就是第 1 章所归纳的伪创新手法“割裂历史”，把参照物拉回到蛮荒时代——如果不用我这个，就什么都没得用。因为还有比我更烂的，所以我就是最好的。

参见：

《潘老师有点理想化了，现实中程序员很菜的占多数》
(umlchina.com/url/lxh.html)

《“以炮换马”的 DDD 歪招是否可以作为起步》
(umlchina.com/url/waizhao.html)

如果有人就是乐意“为爱发电”呢？做出来的东西有没有用，有没有人乐意用都无所谓，那就和本章或本书的内容没关系了，但不得不泼一盆冷水：有权力和有意愿“为爱发电”的人非常稀少。

以上文所说的“钓鱼佬”场景为例，第一个场景中，张三作为一个打工人，没有权力随着自己的喜好“为爱发电”。如果他这样做，就是渎职。第二个场景中，张三也许并不想从钓鱼佬兄弟那里收钱，但他可能追求的是兄弟们的感谢和赞赏，这也是利益，甚至是比金钱还难获得的利益。如果是这样，张三也并非有意愿“为爱发电”。

“为爱发电”也好，想要获利也好，都是个体的自由选择，不分高低。需要批评的是“又当又立”：打着“为爱发电”的旗号，其实却很在乎获利；在乎获利，却懒得去体会目标涉众的感受，以“工程师文化”为名，随性而行。

★有些人的“爱”，其实是自己想怎么爽就怎么来，这是自私自利，根本不是爱他人。

2.3.4.3 因果颠倒和装模作样

定位的时候，装模作样给出了一个结果，但这个结果不含任何思考的价值。

例如，问“这个系统给谁用”，答“给适合用这个系统的人”，这就是正确而无用的废话，但一些人乐此不疲。

很多时候没那么直接，可能会先入为主地设想系统的一个功能 F，然后从这个功能出发来定义目标组织规格——“特别需要 F 来做事情的人”。

例如，一个二手交易的系统，开发人员认为“官方鉴定”是一个亮点，于是把目标组织规格定位为“特别需要二手交易官方鉴定服务的人群”，看起来好像正确，但并未包含任何增值的信息。

我们在业务建模 workflow 所做的一切，目的就是推导出功能“正确”的系统。这里的“正确”，判断标准就是利润。如果先想好了某个系统或系统的某个功能是什么样子，再以此为出发点去“推导”业务建模的内容，这实际上就是因果颠倒。

★伪创新的诱人之处：批量产出“看起来好像正确，但并未包含任何增值的信息”。

我们需要刻意避开先入为主的内容，从其他视角来思考。例如，如果预想了“二手交易官方鉴定”可能是一个亮点（当然，这样的预想不

是凭空产生的,而是建模人员在大脑中朦朦胧胧地做了未知对错的业务建模),定位时就要避开这些字眼,从其他视角思考,如“大学生还是家庭主妇更像目标客户”。

即使不是开发软件系统,而是做其他事情,在这一点上,定位的道理也是类似的。例如开一家餐馆,如果问目标客户是什么人,从“功能”出发回答“来吃饭的人”或“肚子饿的人”没有任何增值信息,并不能帮助推导出餐馆合适的选址、环境风格、菜品。如果先避开餐厅的功能,定位出“科技公司的年轻打工人”,就可以推导出有价值的信息。

在第1章中说过,伪创新擅长“换词”、“砌词”。有时候,废话会经过包装。例如,做一款功能类似于抖音、快手的app,目标组织规格定位为“爱刷一族”或“视觉体验爱好者”,看起来就隐蔽得多,不像“喜欢刷短视频的用户”这样,容易让人一眼看出是废话。

注意,我们并不是要禁止反向思维。从业务建模推导出系统需求后,再把结果放回业务建模的工件来验证结果是否正确,这是可以的,不是因果颠倒。

导致因果颠倒的“验证”是:以所谓的“系统需求”为依据,否定了业务建模的内容。相当于考试做题,也不打草稿理思路(反正也不会),直接就乱答。答完把结果2456代回题目验算,发现不对。此时,应该检查自己的解题思路是否正确,但这太辛苦了!很多做题的人下意识优先怀疑题目有错,要改题。

这也是现实中很多“敏捷试错”受欢迎的原因。如果诚实地“敏(无)捷(脑)试错”,错了就真的老老实实重做,那是非常非常累的。如果

板子真的打到自己身上，“大聪明”们就不会鼓吹“敏捷试错”了。“大聪明”的选择是，把判定错误的尺度隐藏在自己的大脑里，主打一个 by experience、心领神会和拈花微笑。

所以，在一些改起来比较麻烦的地方，他们并不认为自己错了，而是想办法换一个适用的场景——能不能真的“改题”并不要紧，开发人员只需要假设能就行了，反正亏损的又不是他。

★伪创新诱人之处：留出了偷奸耍滑的空间，让懒惰者自行把握判定的尺度。

以为可以随意“换题”的幻觉，还带来了一个有趣的现象。在某些不知柴米油盐贵的开发人员眼里，“做产品”比“做项目”更容易，因为“做产品”看起来没有“做项目”那样难侍候的“客户”或“甲方”，自己想怎么做就怎么做！

2.3.4.4 假装圣母

要做一个电子病历系统卖给医院，说“客户是医院”肯定不够。医院有大型三甲医院，也有二级医院、中心卫生院，甚至社区卫生服务中心……还有根据性别细分的男子医院、女子医院，根据人体器官细分的口腔医院、肝病医院、肛肠医院……还不要忘了国外的医院，美国的 JHH、Mayo、乌干达的@%&，孟加拉的&¥#……。

建模人员使用前文所述的方法定位出目标组织为“北京大兴中医院”。这时，可能就会有人质疑，要是我们只关注“北京大兴中医院”，那“北京协和医院”是不是就漏掉了？难道“北京协和医院”不重要吗？

事实是，这个时候“北京协和医院”确实不重要。“大兴中医院”想要的都还没有满足，第一桶金都没挖到，去想“协和医院”干什么？害怕“漏掉”的想法是幼稚的。需求是一口深井，永远做不完。只要您愿意，可以满世界去调研所有医院，甚至不用调研，拍脑袋就可以得出上万条“需求”。

★上面的故事也可以改成：建模人员定位了“北京协和医院”，有人质疑：难道“北京大兴中医院”不重要吗？

警惕“难道.....不重要吗”的话术。说话人可能在假装圣母，以胸怀天下的姿态有意无意地扩大讨论范围，然后在更大范围内选择对自己更有利的结果。也许是去“北京大兴中医院”调研太累了，扩大范围后挑选自己方便调研的医院或者之前已经调研过的医院，这样可以省时省力。

这样的现象，在软件开发的各个工作流中都会观察到。

例如，针对一个外卖平台，产品经理定位下一步最应该做的工作是优化骑手配送路线。这时，有人质疑“难道其他地方的优化不重要吗”，目的是要扩大范围，然后再落实到和自己利益相关的“优化”上，例如App界面的优化——如果能实现，就会成为自己作品集的一部分。

社会生活中也有类似情形。例如，我们说要保障人的权利。假圣母会说，人只是动物的一种，难道其他动物的权利就不重要吗？于是，范围从人扩大到动物，然后在动物这个大范围内，落实为优先保障自己所养的宠物的权利。

为什么不既要又要都要呢？严格来说是可以的，只不过要排序，因

为资源是有限的，要先解决排序最靠前的问题，再解决排序最靠前的问题，再解决排序最靠前的问题.....

不只是定位目标组织，一直到后续各个工作流的各个步骤，都需要有排序的思想，习惯说“最”，警惕说“也”。这根线如果守不住，会很快滑坡，陷入废话和浪费的泥潭——当然，这也是一些盖着遮羞布的开发人员有意无意想要达到的。

2.3.5 关于定位的其他讨论

2.3.5.1 目标组织可以变化或多个吗

假设目标系统规格为 S1，定位目标组织为 O1。随着 S1 的不断开发，O1 存在的问题不断得到改进。如果存在和 O1 属于同一个组织规格的另一个组织 O2，O2 的负责人希望 S1 也来改进组织 O2 的问题。那么，组织 O1 中在负责人看来排序较高的问题得到改进之后，是否可以把目标组织切换为组织 O2？

上面的说法可能比较抽象，我们把它具体化：

浑元太极科技开发一款“点到为止·餐饮管理系统”，定位目标组织为“化劲火锅”。根据负责人马宝群的观点，“化劲火锅”最需要解决的问题依次为：带有券和储值卡的混合支付更方便、更高效识别高价值客户、生意好的时候备菜更精准.....。

正在针对这些问题依次开发系统功能时，“肾摸事火锅”也想要这个系统，但“肾摸事火锅”负责人认为叫号混乱是“肾摸事火锅”当前最需要解决的问题，而这个问题“化劲火锅”这边并没有专门提到。

此时，是否可以认为照顾“化劲火锅”的利益照顾得差不多了，应切换到“肾摸事火锅”，解决“肾摸事火锅”最头痛的问题？

不可以。

事实上，“化劲火锅”也有排队叫号，但它已经通过其他系统来解决。在“化劲火锅”负责人看来，比起其他的问题，排队叫号的问题实在是不值一提。

可能会有人认为这是“集思广益”、“补漏洞”，觉得幸好多看了一家“肾摸事火锅”，否则就漏掉了“叫号混乱”这么一个“大”问题。

火锅店千千万，“肾摸事火锅”为什么进入了建模人员的视线？稍微认真追溯一下，往往会发现很多时候背后的理由如同儿戏，离得近，有熟人，碰巧看到……

更可怕的是，“肾摸事火锅”有可能是建模人员用潜意识“筛选”出来的。解决了“化劲火锅”排序前三的问题，接下来排第四的问题太难解决了，怎么办？

建模人员的潜意识以自己擅长而且喜欢解决的问题为条件，在众多火锅店中筛选，哎，找到了，对于“肾摸事火锅”来说，该问题最严重。于是，“肾摸事火锅”非常自然地进入了建模人员的视线。

这已经逐渐接近于 2.3.3.1 所说的虚构目标组织，变成了偷奸耍滑的遮羞布。

选择“化劲火锅”作为目标组织，是经过多方比较得到的，如果没有足够的理由，就要尊重当初的选择。如果反复思考，认识到确实是当

初考虑不周，可以更换目标组织，但在更换的同时，需要重新思考之前的排序。

我们再看另外一个场景。

建模人员不是出于前面所述的那些“儿戏”般的原因另外考虑“肾摸事火锅”，而是在定位目标组织时，除了“化劲火锅”之外，多考虑了一个候选的目标组织。这样做虽然也有被用作遮羞布的危险（既然可以多考虑一个，不妨再多考虑两个、三个……），但危害比之前所说的场景要小很多。

下面这个场景看起来相似，但性质不同。

一家企业，当前最值得改进的问题是排程的灵活性，这是生产部的问题，接下来是风险预警，这是供应链部的问题。这时候，不能说切换了目标组织，因为目标组织是这家企业。

当然，如果做的只是“排程系统”，也就不存在什么“接下来是风险预警”的问题。

2.3.5.2 目标组织规格范围的调整

如果目标组织是个人组织，目标组织的范围就是这个人加上围绕这个人的非人系统。

如果目标组织是机构，就需要考虑机构范围多大的问题。例如，范围是 A 公司，还是缩小到 A 公司里面的某个部门，甚至是部门里面的部门，或者扩大到包含 A 公司以及其他子公司在内的集团？

在前面定位的思考中，已经回答了这个问题。从前文提到的“北京

大兴中医院”、“HN 省 HH 市 ZJ**烟叶合作社”、“化劲火锅”等结果，基本可以了解目标组织的边界在哪里。

在接下来的思考步骤中，这个范围可能由于种种原因需要调整。原因可能有：

(1) 目标组织负责人和目标组织不匹配。

例如，目标组织负责人是 A 医院科研处处长，目标组织是 A 医院。

(2) 发现改进目标和目标组织范围不匹配。

例如，目标组织是 A 医院，但改进目标只围绕科研，如“降低立项和审批的人力成本”。

(3) 在建模业务流程时发现目标组织的大多数流程和目标系统要引入的改进不相干。

例如，目标系统规格只和医院科研流程相关，目标组织却设定为 A 医院。

(4) 在建模业务流程时发现目标系统要改进的很多流程不在目标组织范围内。

例如，目标组织设定为 A 医院科研处，但目标系统要改进的流程大多和经费相关，更值得研究的是 A 医院财务部门。

注意两点：

(1) 严格来说，不存在什么范围调整，而是修改了目标组织。

(2) 只是说可能要调整。是不是一定要调整，还需要结合后文的知

识来考虑。

这个范围调整不只是目标组织的范围调整，而应该看作目标组织规格的范围调整。调整之后，在条件允许的情况下，需要重新做前面的定位思考。

例如，一开始定位目标组织为 A 医院，后来发现范围太大，调整为 A 医院科研处。这个情况，应该看作把目标组织规格从“医院”改为“医院科研处”。此时，需要重新做定位思考。

前面的定位思考中，A 医院从众多医院中脱颖而出，成为目标系统规格最合适改进的医院，不代表接下来的定位思考中，A 医院科研处还能继续从众多医院科研处中脱颖而出，成为目标系统规格最合适改进的医院科研处。

如果为时已晚，目标系统规格已经和 A 医院深度绑定，那就是本书方法学之外的问题了。

2.3.5.3 关于定位目标组织负责人的错误

首先要说明的是，以下归纳的错误，有一部分原因是过去用“老大”一词来指代目标组织负责人导致的。如果不再使用“老大”，而是使用“目标组织负责人”，类似的错误会大大减少。

“老大”用于指代目标组织负责人，最开始是我在备课时造的词，后来写在书中，已经用了 20 多年。我收到的关于我的课程和书籍的反馈中，有不少学生和读者说“印象最深的就是要找对老大”，可见造词的力量。问题是，学生和读者记住了“要找对老大”，但他们理解的“老

大”和我讲的不是一个意思！

因此，我决定放弃“老大”这个已经在读者群体深入人心的词，改用传播力量不那么强而且字数又多的“目标组织负责人”。

错误一：把机构的 IT 主管当作目标组织负责人

在目标组织是机构的情况下，很多时候，建模人员并不能经常见到真正的机构负责人。他经常见到的干部可能是该机构的 IT 部门主管，头衔可能是“信息中心主任”或“信息处处长”之类。这个 IT 主管平时从事的也是计算机或软件方面的工作，双方一见面，大谈特谈“架构、大语言模型、智能体……”，业务方面却难得深入讨论。

如果目标组织就是 IT 部门，那么 IT 主管作为目标组织负责人是合理的，但往往实际情况并不是这样。实际情况是，系统要改进的是整个组织的流程或者业务部门的流程，建模人员却把经常见到的干部——IT 主管当作目标组织负责人，这是不合适的。

注意，以上只是方法学上的道理。实际的交流中，不妨碍建模人员口头上把 IT 主管当成目标组织负责人来吹捧，只是心里要牢记他不是组织负责人，和他交流的目的是通过他去揣摩真正的组织负责人的想法。

用看病类比，在患者病情严重或者患者不便交流的情况下，和医院的医生频繁打交道的可能是患者的家属。有的患者条件好，雇佣有专职的家庭医生，由家庭医生来介绍情况。

不管是家属还是家庭医生，他们只是比较熟悉患者情况的人，不是患者。医院的医生切不可因为他们好打交道或打交道较多，就让家属或

家庭医生上检查床或手术台。

错误二：把过大的大领导当作目标组织负责人

例如，要做一个系统改进某大型电商企业仓储部门的拣货效率，把目标组织负责人定为该电商集团公司董事长，很可能是不合适的。董事长关注的是整个组织的指标，而不是某个系统给某个小局部带来的改进。董事长关注的指标，可能适用于成百上千个系统的引进，无法直接指导建模人员寻找和过滤特定系统的需求。

这样的错误在使用“老大”一词的时候，是比较容易出现的，毕竟董事长最大嘛。如果改用“目标组织负责人”，通过前面的定位思考，得到目标组织是仓储部门，自然而然能推出目标组织负责人是仓储部门的负责人（头衔为仓储总监或供应链总监之类）。

此时，再冒出一个董事长来抢走目标组织负责人的位置，这是很少见的。如果还是发生，原因已经和“老大”一词的误导无关。有可能是董事长在视察工作时，对仓储部门的某台自动拣货机很感兴趣，还即兴发表了一些意见，让人误以为“自动拣货机”这个系统的“老大”提升到董事长了（好吧，还是“老大”一词在作怪）。

董事长不是“自动拣货机”的目标组织负责人，他不会真的对“自动拣货机”有什么指标，他关注的是集团的指标，而这些指标适用于指导集团的方方面面，包括“自动拣货机”。建模人员可以从董事长的意见提炼出董事长所关注的层面上的指标，确保“自动拣货机”给目标组织带来的变化至少不会和董事长关注的指标背离。

更进一步说，假设国家领导人到该电商企业的仓储部门视察，饶有

兴趣地对着自动拣货机和工程师亲切交流，并发表重要讲话。此时，也要从领导人关注的层面上领会其中的精神，而不是简单认为领导人说的就是这个小东西、这个部门和这个企业。

错误三：认为出钱的人是目标组织负责人

改进的资金有各种各样的来源，不能说谁出钱最多谁就是老大（还是“老大”一词在作怪）。出资的人的利益，要由目标组织负责人综合考虑，通过他来体现。

还是以看病类比。患者治病的钱可能是自己出，也可能是家属出，政府出，同房病友捐赠，甚至由医院买单。不管怎样，病灶在患者身上，该上手术台的是患者，这与有钱没钱无关。

以上只是从普遍的意义归纳了一些典型的错误，但并没有说不允许有特例。

IT 主管就不能是组织负责人吗？没准这家公司的法定代表人只是傀儡和背锅侠，那个表面是 IT 主管的人才是实际控制人。目标组织是部门时，也会存在部门一把手被副手架空的情况。

此时，建模人员要定位到真正的组织负责人，但这和警惕上面所归纳的错误不矛盾。

2.3.6 自测题



本书不提供习题答案，请扫码或访问

umlchina.com/book/quiz2_1.html 完成在线测试，做到全对，自然就知道答案了。



1 [单选]

以下是某三甲医院肿瘤科病房护士某天的工作描述。

7:20-7:40

更衣、晨会交班：听取夜班情况

7:40-8:20

查房准备：核对当日医嘱清单、化疗/靶向/免疫用药安排，整理输液泵、PICC 维护用品、止吐/预处理药物

8:20-9:40

*给多位患者做静脉输液/连接输液泵

*核对化疗药、稀释液、滴速与给药顺序

*处理恶心、过敏、输液外渗等不适

*同步在信息系统里逐项确认/记录执行时间

9:40-10:10

采血/标本交接：按床号贴条码、核对身份、送检登记

10:10-11:30

继续执行医嘱

患者/家属咨询

抽空补录刚才没来得及点完的执行记录

11:30-13:00

午餐、短暂休息

13:00-14:30

把上午化疗执行记录、输液泵参数、患者不良反应、生命体征、护理评估等从纸上/记事本一条条录入信息系统

.....

如果做一个信息系统进一步改善该护士的工作,这个系统最应该提供的功能是_____。

- ☐A) 把化疗方案和给药顺序投到床旁大屏
- ☐B) 扫一下药袋/输液泵/腕带条码获得相应信息
- ☐C) 统计科室化疗人次、不良反应发生率与用药分布
- ☐D) 信息不足, 看不出来

2 [多选]

前些年“敏捷”是热词的时候,出现了许多“年轻资深敏捷咨询师”。

他们到处演讲、造词、指点江山。张三是其中的一员。

以下说法正确的有：

- ☐A) “年轻资深敏捷咨询师” 可以看作一个组织。
- ☐B) “年轻资深敏捷咨询师” 可以看作一个组织规格。
- ☐C) 张三可以看作一个系统规格。
- ☐D) “年轻资深敏捷咨询师” 可以看作一个系统规格。

3 [单选]

以下可以作为目标组织负责人的是：

- ☐A) 居住在美国纽约法拉盛的中国公众人物罗玉凤
- ☐B) 世界五百强之一的零售商家乐福公司售货员
- ☒C) 美利坚合众国参议院议员
- ☐D) 法拉盛大街 43 号大东北饭馆服务员

4 [单选]

研发部急需一名大模型算法工程师来优化公司所开发系统的 AI 能力,由人力资源部负责出面招人。以这名大模型算法工程师为研究对象,目标组织负责人是：

- A) 人力资源部经理
- B) 研发部经理
- C) 公司总经理

D) 大模型算法工程师

5 [单选]

以下选项中哪一类开发组织的负责人不需要思考“定位目标组织”的问题？

☐A)处于苟活状态，靠人脉接点小项目维持的开发组织

☐B)实力在业界排名上游的开发组织

☐C)有雄心壮志做产品的开发组织

☐D)都需要思考

6 [单选]

有开发团队想做一款 AI 视频生成工具，起名 Sorb。如果问“这款产品给谁用”，以下回答中最有某些领域驱动设计伪创新“投资少、见效快、产量高”风格的是：

☐A) 人

☐B) 想用 Sorb 做视频的人

☐C) 二次元创意工作者

☐D) 需要快速制作场景原型的独立电影人

7 [单选]

请把左侧信息系统和右侧目标组织画线对应。

1 领英(LinkedIn)

a 二次元爱好

者大学生嘉豪

2 小红书

b 精致生活

分享者易孟菱

3 哔哩哔哩

c 某大厂 P8

马宝民

A) 1-a, 2-b, 3-c

B) 1-a, 2-c, 3-b

C) 1-b, 2-a, 3-c

D) 1-b, 2-c, 3-a

E) 1-c, 2-a, 3-b

F) 1-c, 2-b, 3-a

8 [单选]

为了在定位目标组织时尽量不犯“吃窝边草”的错误,讨论问题时,我们要特别警惕言辞中的某些词(字),例如_____。

☐A) 最

☐B) 也

☐C) 效率

☐D) 老大

9 [多选]

以下场景中，针对“系统 X”而言，开发组织和目标组织可以看作是同一个组织的有：

☐A)太极银行的信息技术部开发了“系统 X”，用于改进信息技术部的“服务器部署和故障响应”流程。

☐B)太极银行的信息技术部开发了“系统 X”，用于改进零售信贷部的“信用卡申请审批”流程。

☐C)太极银行的信息技术部开发了“系统 X”，用于改进太极银行全行的“内部公文流转和审批”流程。

☐D)太极银行的信息技术部开发了“系统 X”，吸引二次元年轻人来使用。

10 [单选]

63 岁的马大爷找到“和为贵牙膏”公司，跟产品负责人说：

“我用你们家牙膏 30 年了，尤其是夏天刷完牙那种薄荷凉到天灵盖、口腔像开了空调的感觉，太爽了！不过我现在年纪大了，牙龈敏感、牙本质敏感，再来这种‘强刺激薄荷’就受不了，刷完还会刺痛。

所以希望你们在保持原来优点的同时，照顾一下老年人。我这里有一个方案：用微胶囊/纳米包裹技术把清凉物质包起来——入口先释放一小部分产生‘爆凉’的爽感，但总体刺激性更低；再把摩擦剂颗粒优化得更温和，做到‘爽’但不‘辣口、刺牙’……”

“和为贵牙膏”公司的产品负责人大概率会先感谢马大爷的精彩建议，然后置之不理。原因有可能是：

- A) 目前热爱“越凉越狠”重口味的年轻人基本盘还比较厚实
- B) “和为贵牙膏”公司老了，无法和顾客一起成长
- C) “和为贵牙膏”公司产品负责人没有领悟革命性创造和划时代洞见的领域驱动设计背后的敏捷禅意
- D) 微胶囊/纳米包裹技术太深奥，“和为贵牙膏”公司没有这个技术力量

11 [单选]

研究对象为一款和医院相关的信息系统。在定位目标组织的过程中，建模人员用“爆炸法”询问，对方回答“中山大学附属肿瘤医院”。

这时，建模人员继续提问的问题最应该是哪一个：

- ☐A)附近建设六马路有一家“YMY”餐厅，你去吃过吗？
- ☐B)这个系统涉及到整个医院还是仅限于医院的某个部门？
- ☐C)你认为是中山大学附属肿瘤医院的最主要理由是什么？
- ☐D)院长的愿景是什么？

12 [单选]

如果我们把建模愿景的步骤进一步细化成：

- 1 思考开发组织的能力
- 2 从开发组织能力定位目标组织规格
- 3 从目标组织规格定位目标组织

4 从目标组织定位目标组织负责人

5

以上哪一个步骤,如果 AI 详细回答了,最有可能会引起法律纠纷?

☐A) 步骤 1

☐B) 步骤 2

☐C) 步骤 3

☐D) 步骤 4

13 [多选]

关于 Sparx Systems 开发并出售的建模工具 Enterprise Architect (简称 EA), 以下说法中错误的有:

☐ A) 如果以 EA 系统为研究对象, 系统的目标组织是 Sparx Systems 公司的 EA 研发团队。

☐ B) 当 Sparx Systems 公司的 EA 研发团队也开始使用 EA 时, Sparx Systems 公司的 EA 研发团队就会成为 EA 的目标组织。

☐ C) 当有其他公司想研发一款建模工具 (代号 X) 来和 EA 竞争时, Sparx Systems 公司就会成为 X 的目标组织。

☐ D) 对 EA 系统来说, Sparx Systems 公司的 EA 研发团队成员是非常重要的涉众。

2.4 建模步骤 A-2 定位系统的愿景

2.4.1 愿景的定义

如果有一个系统规格 S ， S 的开发组织是 $O1$ ，开发组织负责人是 $P1$ ， S 的目标组织是 $O2$ ，目标组织负责人是 $P2$ ，那么系统规格 S 的愿景定义为：

在 $P2$ 期望的 $O2$ 的改进期望集合中， $P1$ 认为 S 能满足且当前排序最高的那一项。

把这些概念添加到之前已有的类图上，得到类图如图 2-30：

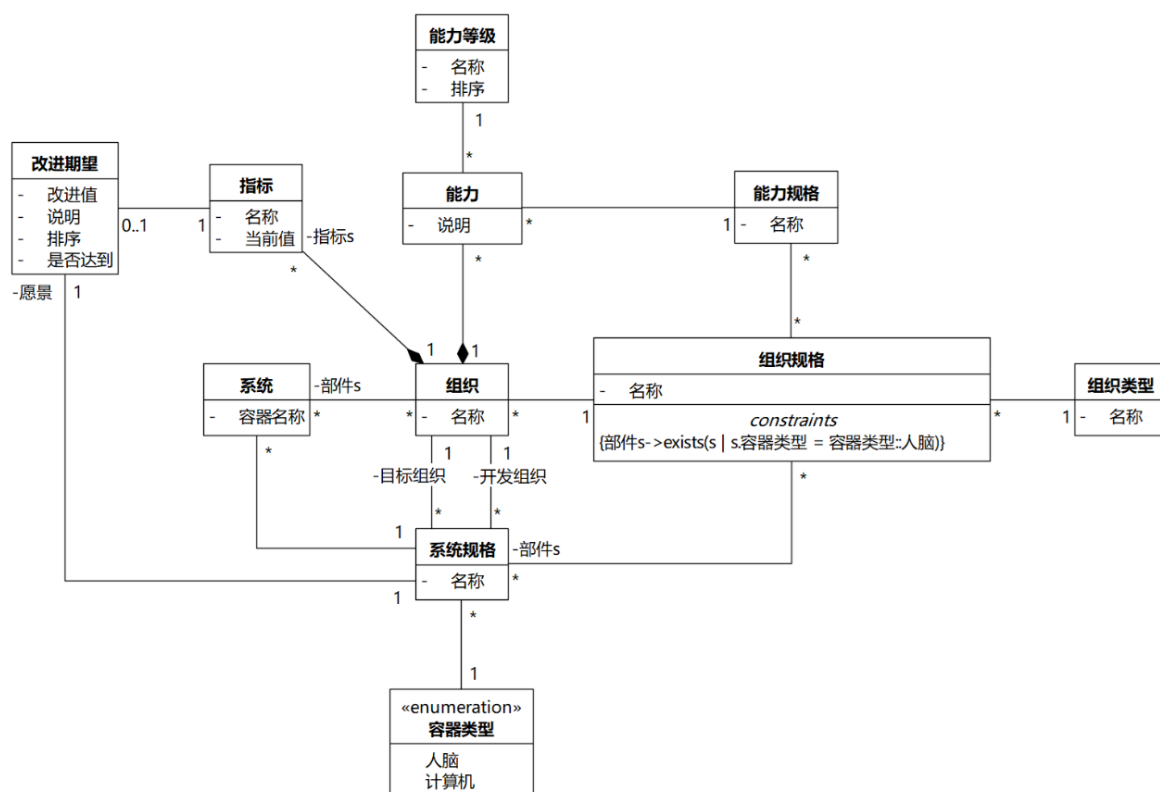


图 2-30 添加了愿景定义相关概念之后的类图

“系统规格”和“改进期望”之间的关联不是冗余的，因为从目标

组织的若干改进期望中挑选哪一个作为愿景,取决于建模人员所揣摩的开发组织负责人（P1）的选择。也就是说，从“系统规格”出发，沿着图 2-31 所示的路径，无法推导出愿景。

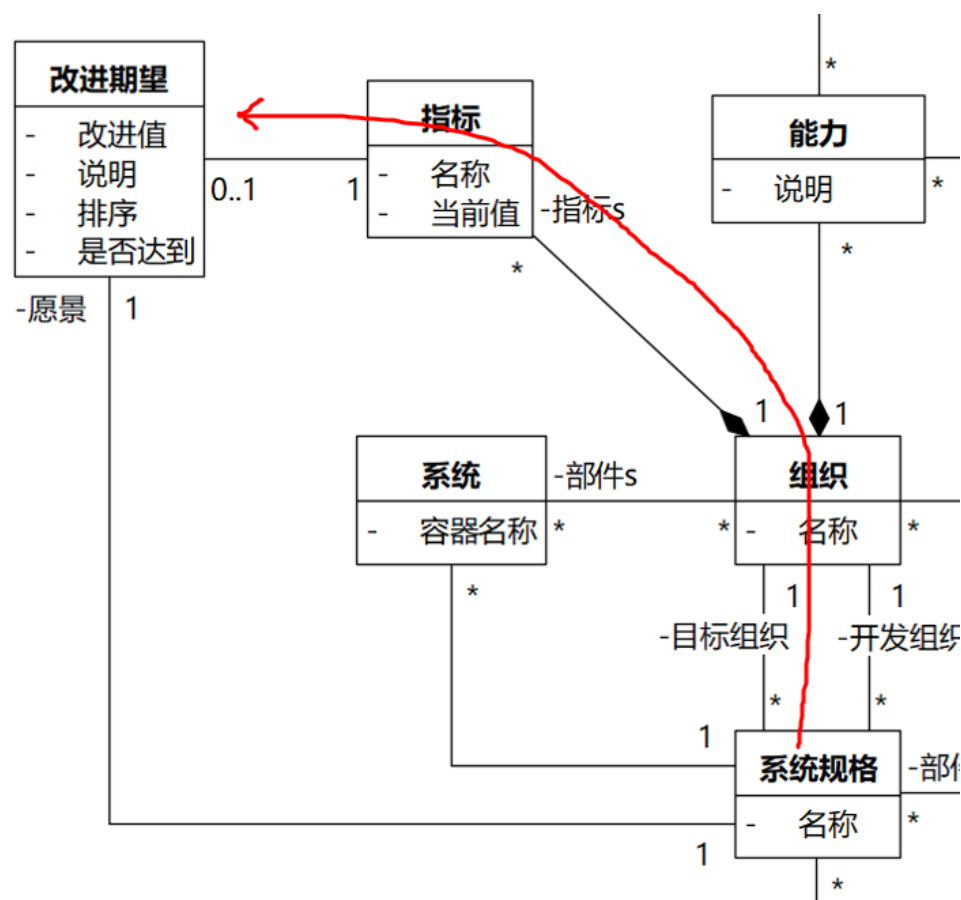


图 2-31 按照这条路径无法推导出愿景

本小节开头的文字，用对象图表示如图 2-32：

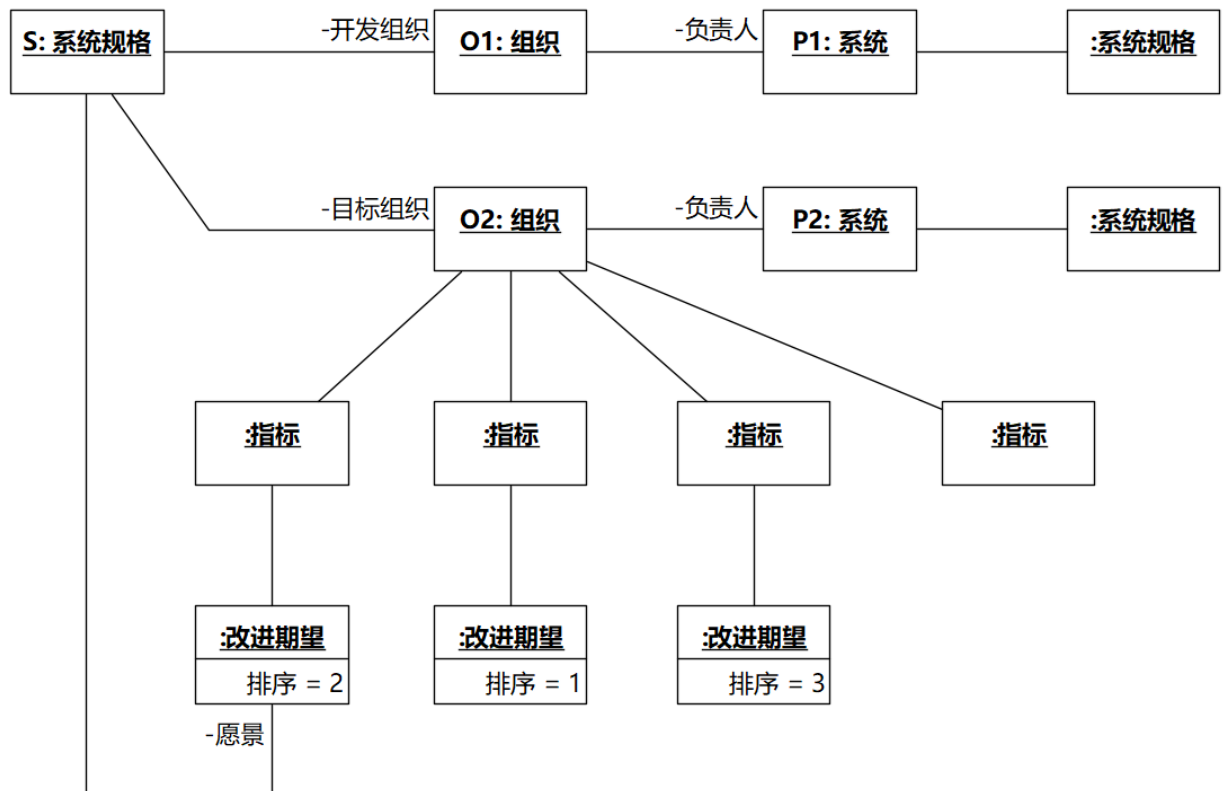


图 2-32 本小节开头的文字的对象图

我们可以把 S、O1、O2、P1、P2 替换成更接近真实的内容，以帮助理解：

S（系统规格）：四两拨千斤系统

O1（开发组织）：很快啊供应链科技有限公司

P1（开发组织负责人）：很快啊供应链科技有限公司总经理 马宝
国

O2（目标组织）：京西集团华东一号转运中心

P2（目标组织负责人）：京西集团运营总监 刘劲西

刘劲西刘总（P2）对京西集团华东一号转运中心（O2）有如下改进期望：

*提升单位面积的货物存储量（排序=1）

*降低大促期间包裹的分拣错误率（排序=2）

*降低仓库电费支出（排序=3）

马宝国马总（P1）认为“降低大促期间包裹的分拣错误率”是四两拨千斤系统（S）的愿景。

对象图如图 2-33：

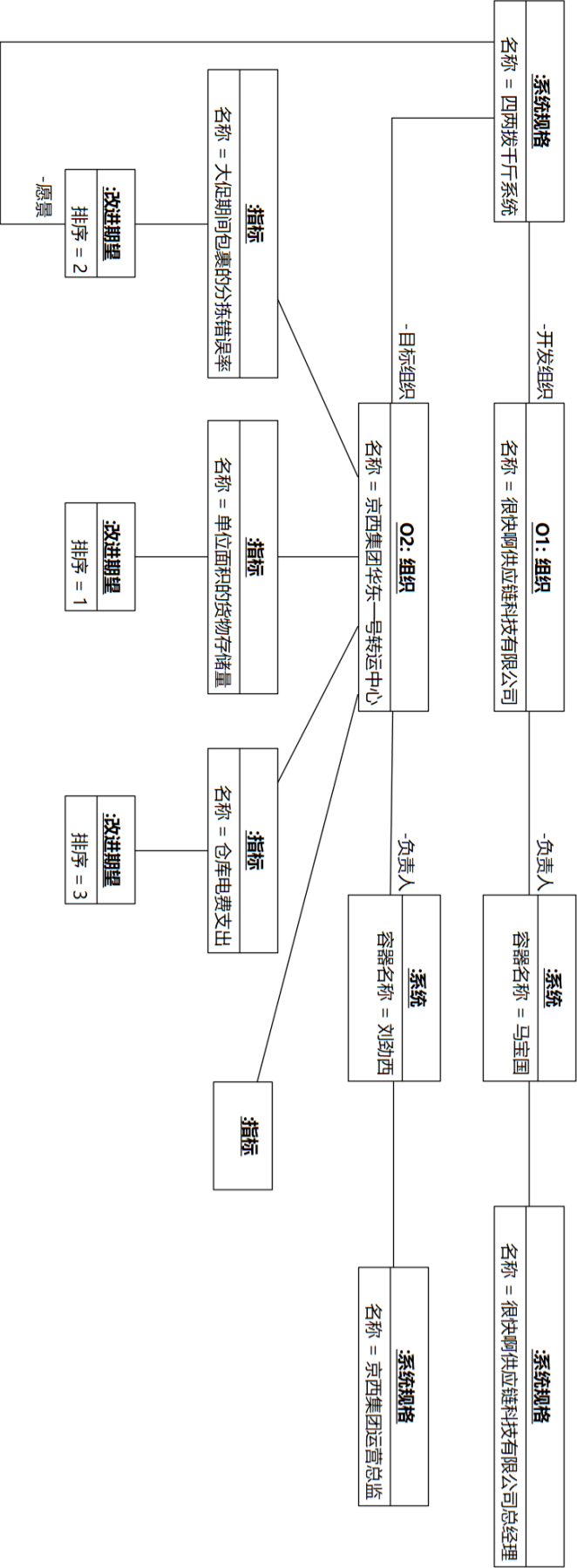


图 2-33 供应链领域的对象图

可以在“系统规格”上加一个约束，确保愿景来自目标组织（的目标集合）的改进期望集合。添加了约束后的类图如图 2-34：

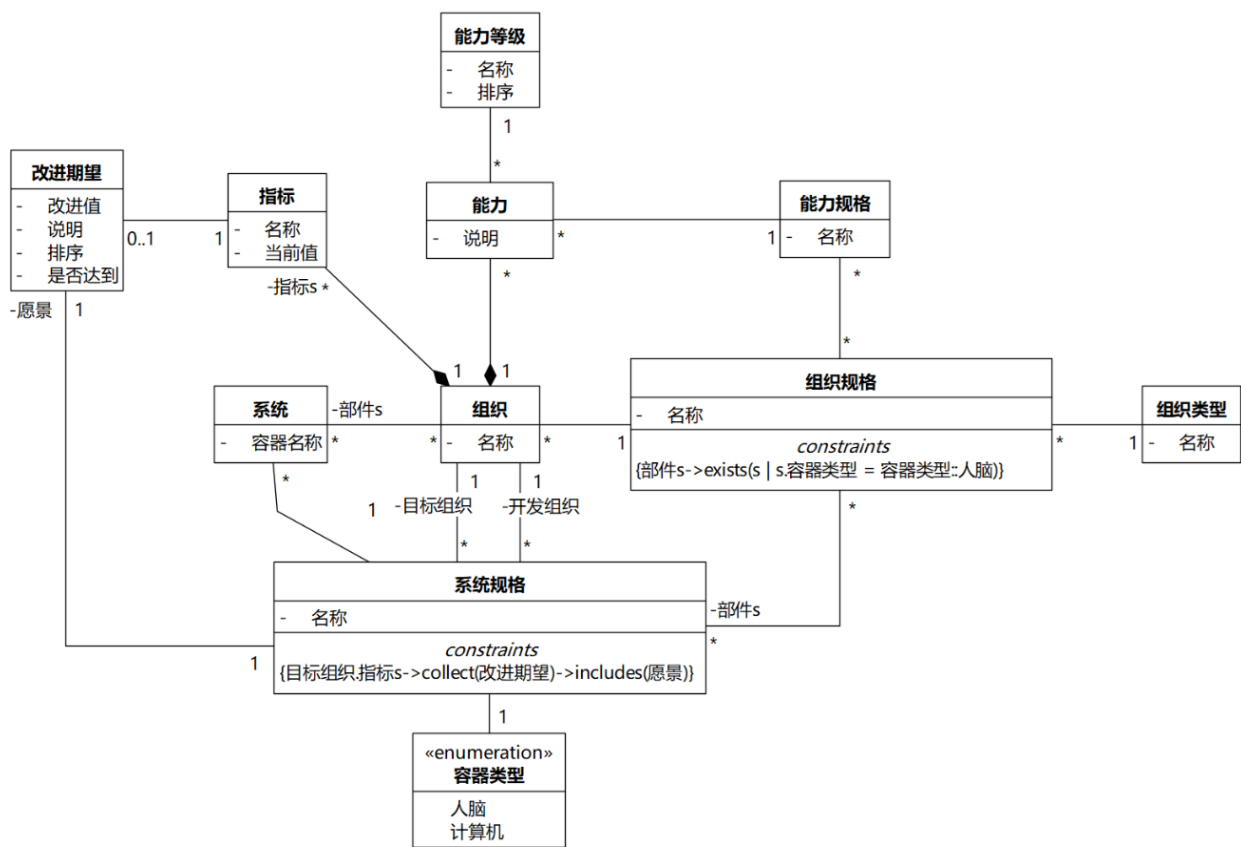


图 2-34 加上约束后的愿景相关概念类图

OCL 支持把 `aSet->collect(name)` 简写为 `aSet.name`，图 2-34 中约束也可以改为：目标组织.指标 s.改进期望->includes(愿景)。

如果违背了这个约束，例如，马宝国认为的愿景并不在刘劲西在意的改进期望之中，应该怎么做？

(1) 马宝国想办法“掰正”刘劲西的改进期望。

这个难度太大，而且如果有这个能力，做其他的也许回报更高。

(2) 马宝国改变想法，从刘劲西在意的改进期望之中选择愿景。

这个做法至少是从刘劲西的视角考虑问题的，算是成功了一半，另一半就是能不能竞争得过和自己有类似想法的牛宝国、杨宝国了。

(3) 马宝国怀疑刘劲西不是正确的目标组织负责人，重启前文的定位思考。

这个做法是最正确的，但风险在于马宝国此时可能会有“累了，不想努力了”的想法，所谓“重启定位思考”压根就没有走前文所说的思考过程，而且随便换了一个能迎合自己想法的雷劲西或张劲西了事。

关于图 2-34 的一些说明：

图 2-34 中，我们把指标的名称和当前值放在一起，并未分离出一个和“组织规格”关联的“指标规格”，然后把“指标”关联到“指标规格”。

这样做的主要原因是难以归纳和统一。即使两个组织属于同一组织规格，很可能两个组织的负责人认为是指标的东西有不少差异。在研究一个新的目标组织时，如果想复用（抄袭）之前的组织的指标，可以通过统计和该组织属于同一组织规格的组织的指标得到。

另外，图 2-34 中“系统”有一个属性“容器名称”，这是如[2.1.3 系统和系统规格]所说，在模型中省略了“容器”，然后把“容器”的“名称”放入“系统”中，如图 2-35 所示。

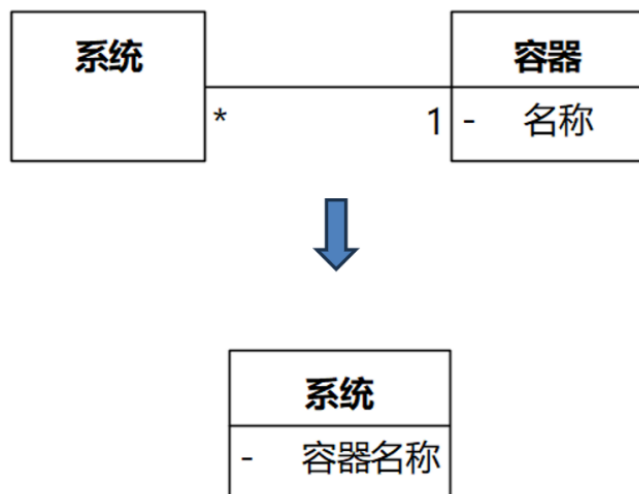


图 2-35 把“容器”合并入“系统”

后文的[8.2.5.2 属性是否直接描述类]详细评价了把一个类合并到另一个类的做法。

这时，如果“刘劲西”在组织中身兼数职，而且模型需要关注，那么会出现多个“系统”的“容器名称”都叫“刘劲西”，而且无法辨别这些“刘劲西”是同一个人（容器）还是同名但不同人。不过，出现多个“刘劲西”的情况不会很多，而且从本书方法学的视角，无法辨别是否同一人，不影响建模的推导。

2.4.2 愿景的要点

2.4.2.1 改进期望，不是系统功能

系统的愿景是系统能满足的组织指标的改进期望，而不是“系统能做某事（系统的功能）”。请比较图 2-36 左右两列的内容。

像愿景的表述	像功能的表述
提高营销文案与用户画像	能够利用大模型生成营销文

的匹配度	案
提高供应链碳足迹数据覆盖率	建设供应链碳足迹追踪与核算系统
缩短车险小额理赔案件处理周期	拍照并通过 AI 识别损毁程度自动定损

图 2-36 改进组织指标，不是做某事

愿景和功能是不是——对应的。一个愿景的达成可能需要在组织流程中使用系统的多个功能。例如，图 2-37 中，“提高防汛决策准确度”是愿景，但不是系统功能。系统没有提供这样一项功能，领导输入一个准确度数值，确认，防汛决策准确度“duang”的一下就提高了。只有在各个岗位分别使用系统的“查看云图”、“上报水库运行情况”等功能之后，组织的“防汛决策准确度”这个指标才能得到提高。

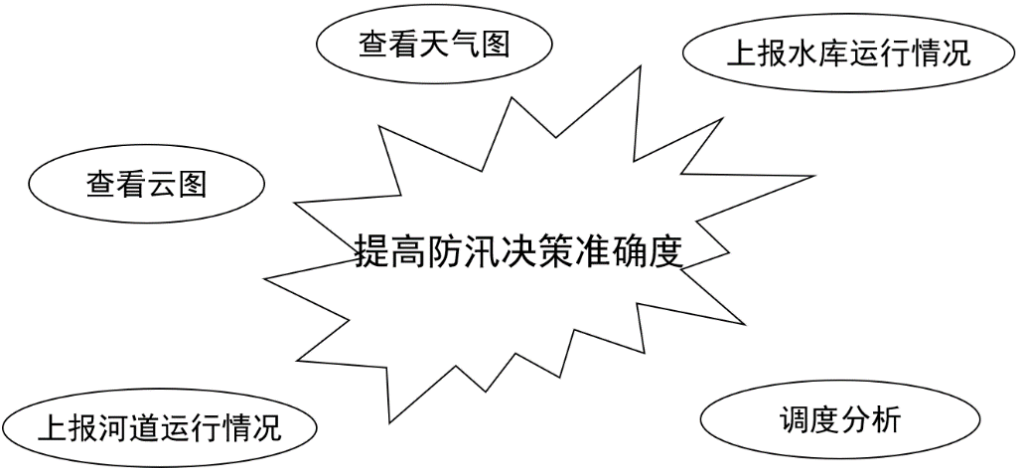


图 2-37 愿景 vs. 功能

一个系统功能也可能会覆盖多个愿景。在达成第一个愿景时，我们

可能得到了系统的一些功能。接下来，达成第二、第三个愿景时，有可能会发现之前的功能已经可以满足。

例如，某工厂可能有一个愿景“降低因电机过热导致的非计划停机率”，带来的系统功能是（给设备增加传感器）实时监控设备温度，超出设定边界则提醒相关人员判断和处理。功能很简单，但目标组织负责人已经觉得可以了，接下来是第二个愿景“降低设备在非生产时段产生的电力损耗”，之前的实时监控设备温度这个功能也能满足。

2.4.2.2 提炼指标不容易

张嘴就说系统有某某功能（图 2-36 右侧）是容易的，想到什么说什么。要说出愿景这样一个可度量的指标改进（图 2-36 左侧）就难一些，但也不是不可以编造。

可以先拍脑袋决定系统有某某功能，再从此出发，推导出这个功能可能会带来什么好处。能做到这一步，已经算是有一定的思考（造假）能力了，可惜，这还是第 1 章所批评的思维颠倒，不能得到正确答案。

我们要思考的是：目标组织负责人在意的是哪些指标？我们的目标系统能改进其中的哪些？揣摩和提炼目标组织负责人在意的指标，是一件困难和辛苦的事情。

如果目标组织是机构，目标组织负责人可能是企业管理人员或政府官员，他们的日程安排很紧张，建模人员不可能经常接触到。如前文所说，代表机构和建模人员经常接触的接口人很可能是“甲方信息中心主任”这样的人。这时，需要通过或绕过接口人去揣摩真正负责人的意思。

就算有机会接触到机构负责人,可能会发现他说的话不但比较“艺术”,而且经常“既要.....又要.....”,让人无所适从。这是他用来展示控制权的方式,也是辨别手下人是否和自己一条心的手段。建模人员依然需要通过各种手段揣摩机构负责人云雾缭绕的话语背后的指标。可以阅读负责人的其他讲话、报告,咨询负责人派来的接口人,也可以求助本方高层——同样的一句话,小兵思维层次不够高,觉得云里雾里,也许高层早已洞若观火了。

如果目标组织是个人组织,大多数情况下目标组织负责人(也就是个人组织中那个惟一的人)还是比较好接触到的。例如,要做一款面向屌丝的“黑米”手机,定位了一位屌丝中的屌丝作为目标组织负责人。一顿烧烤,应该足以让他尽吐真言。

(★如果为天皇巨星做一个专用 app,和建模人员打交道的可能是经纪人或助理。)

不过事情依然困难。这位屌丝中的屌丝虽然容易亲近,却没有能力清楚地表达自己存在的问题和改进目标。这时,同样需要建模人员认真去倾听和观察,体察和揣摩他的痛苦。

这些揣摩技能,我们每个人都有。我们每天都在揣摩上司、同事、配偶的意思,只不过现在要把这项技能用在软件开发上。

我们并不能假设负责人在意的指标都是高尚的。有时建模人员抱怨“哪有什么度量指标啊,我们做的那个系统就是个政绩工程,客户领导去××省考察回来,说××省有一套啥啥系统,我们也要有!”其实,“政绩”二字本身就隐含着数字的概念,政绩工程也一样有指标。建模人员

仍然需要好好去揣摩：目标组织负责人关心什么样的政绩指标？

2.4.2.3 提炼指标的方法

建模人员在和涉众交流时，可以用以下方法帮助提炼指标。

针对形容词思考

找到陈述中的形容词，询问符合和不符合这个形容词的情况。

例如，涉众的回答里有个形容词“规范”，建模人员可以问：目前怎么个不规范，请举一个目标组织负责人最头痛的例子？改进之后，达到什么样的情况，目标组织负责人会觉得规范了？通过这样追问，得到“规范”的指标可能是“格式不合格的报表所占的比例”。

类似的提炼可能还有：

方便——填写一份申请的平均操作次数

高效——从受理到发证的时间周期

从初步设想的解决方案倒推

如果没有这个解决方案，涉众要付出什么代价？

（可以类比前些天流行的“能咋地”）



例如，初步的解决方案是做一个手机上审批费用申请的 app，可以反过来问：

--如果没有这个 app，可能会怎样？

--领导不在办公室时不方便审批。

指标可能是“领导不在办公室期间的平均审批周期”，这是领导在意的。手机 app 只是目前比较适合解决方案。要是能打个响指，空中飘来一朵云，领导在云上划拉两下就行，谁耐烦带个手机弄得鼓鼓囊囊的呢？



借鉴机构的 KPI（关键绩效指标）

很多机构都归纳了自己的 KPI，借鉴过来就可以，如图 2-38 所示：

工艺管理部关键绩效考核指标

序号	KPI 指标	考核周期	指标定义/公式
1	新产品工艺设计 任务完成准时率	季/年度	$\frac{\text{实际设计周期}}{\text{计划设计周期}} \times 100\%$
2	工艺试验 及时完成率	月/季/年度	$\frac{\text{按时完成工艺试验次数}}{\text{工艺试验总次数}} \times 100\%$

图 2-38 某部门关键绩效指标示例

2.4.2.4 典型的指标错误

从愿景的定义以及类图 2-34，我们知道，愿景是：

- (1) 目标组织负责人对目标组织指标的改进期望
- (2) 改进是目标系统能带来的

违反这两个要点之一，必定会使愿景变成废话、套话。当然，这也是伪创新买家和卖家有意无意要达到的。

错误一：把开发组织负责人的“许愿”当成愿景

这是违反了 (1)，用开发组织负责人的期望代替了目标组织负责人的期望。

这样的“愿景”通常表现为开发组织负责人的“许愿”，例如：

*一年以内会员达到一千万

*系统的市场占有率至少达到 40%

会员越多越好，市场占有率越高越好，许个愿的事情谁都会。但你说开会员就开会员啊？你的东西好在哪里，有什么理由让人家不在别的地方开会员，偏偏在你这里开会员？

这就需要建模人员观察和建模目标组织,并揣摩目标组织负责人的心意,找到目标系统可以改进的指标。这个工作很辛苦,建模人员为了逃避,不知不觉就把愿景替换成开发组织的“许愿”。

上面提到“会员”、“市场占有率”,似乎意味着非定制的“产品”才会有这个问题,其实不然。

在定制的“项目”中,建模人员为了逃避,也会简单粗暴地把“愿景”写成“用目标系统做某事”。这样的事情,在现状业务流程中不存在与目标系统相似的系统时更容易发生。

我在本书的前面两版提到一个错例(发生在2005年之前,注意这个信息),某信贷风险系统的愿景被写成“银行风险部能够(用目标系统)对贷款做风险评估”。显然,这属于图2-36的表格右侧所说的“做某事”,而不是左侧的“改进某个指标”。

如果风险评估的现状业务流程中已经使用了一个名为“A 信贷风险系统”的信息系统,只是能力较弱,想要更换成“Y 信贷风险系统”,那么,建模人员厚着脸皮写成“银行风险部能够对贷款做风险评估”的可能性比较小;如果风险评估的现状业务流程中不存在“信贷风险系统”这样的系统,而是由工作人员“手工”(只是建模人员这么认为,其实在用电子表格,这也是信息系统)完成,建模人员厚着脸皮写成“银行风险部能够对贷款做风险评估”的可能性就会大增。

这样的错误直接把目标组织的指标改进和开发组织建造的特定系统绑定,不客气点说,就是把开发组织的利益YY到目标组织身上。

目标组织负责人要的可能是“缩短评估贷款风险的周期”,至于引

入 A 系统、T 系统还是 Y 系统来改进，并没有特定的要求，任由它们去竞争。在竞争中，Y 系统胜出了，Y 系统的建模人员于是就 YY “目标组织负责人的愿景就是要上 Y 系统”。其实，如果是 A 系统胜出，目标组织负责人也会接受 A 系统。当然，A 系统的建模人员可能也会 YY “目标组织负责人的愿景就是要上 A 系统”。

可以用前文所说的“爆炸法”来检验。你向目标组织负责人（俗称“客户”）推销目标系统，话术是“你买了我们这个系统，我们的市场占有率会增加噢！”——他不扇你一个大嘴巴算客气的。

错误二：改进不是系统能直接带来的

有的建模人员“大彻大悟”，认为不管什么系统都是万变不离其宗，目标无非是更多的金钱、更高的地位或更好的名声，就像《史记》所说：天下熙熙，皆为利来；天下攘攘，皆为利往。

没错，最终目标是升官发财出名，但这个目标是“正确而无用的废话”，对于探索目标系统的需求没有帮助。

目前赚钱还不够多的原因可能有很多条，每条原因又有各自的若干原因。目标系统能解决的仅仅是“赚钱少”这个大问题的原因的原因的原因中的一到两条——建模人员必须定位到这里。

例如，“移动病区护士系统”的愿景一开始如图 2-39 所示：

系统：

移动病区护士系统

目标组织负责人:

Z 大学附属第*医院院长 张**

愿景:

减少医疗事故

图 2-39 过大的愿景

“减少医疗事故”这个愿景过大了,适用于各种各样的改进方案(包括更换另一个品牌的手术器械),没有“移动病区护士系统”特有的味道。改为图 2-40 的内容更合适。

系统:

移动病区护士系统

目标组织负责人:

Z 大学附属第*医院病区护士长 李**

愿景:

减少错误执行医嘱事件的发生率

图 2-40 恰当的愿景

如何定位系统能带来的恰如其分的改进,可以使用管理学中的鱼骨图(或称石川图、因果图)来帮忙,如图 2-41 所示:

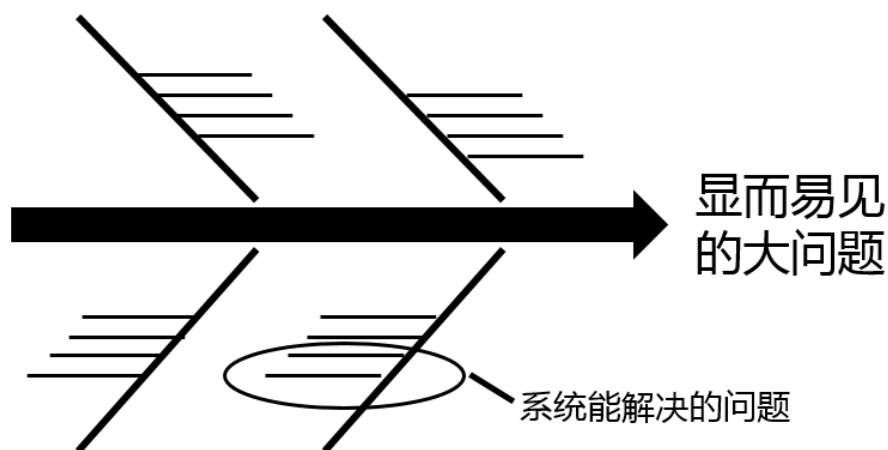


图 2-41 鱼骨图帮助定位真正的问题

UML 中没有鱼骨图，可以用类图代替，如图 2-42 所示：

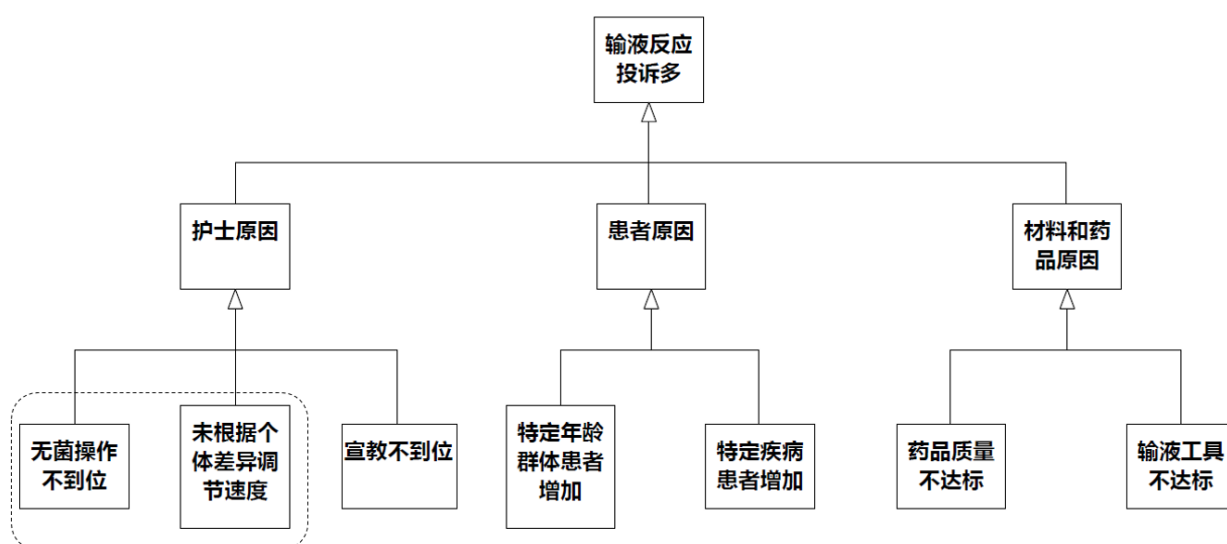


图 2-42 类图代替鱼骨图

在画鱼骨图的时候，同样要警惕先入为主的思维。

有的建模人员在画鱼骨图时，不能客观地探索和列举可能的原因，只列举与自己所臆想的“目标系统”相关的原因——相当于把图 2-42 的患者原因和材料药品原因砍掉，只保留左侧的护士原因。

建模人员之所以这样，可能是能力不足，可能是懒惰，还有一个可

能是害怕，害怕自己之前的臆想被推翻。

如果像图 2-42 这样列出来之后，悲哀地发现，整个社会正在老龄化而且短期无法改善，即使引进目标系统能改进左侧圈起来的问题，在整个大指标恶化的形势下，也变得毫无意义。这时就需要建模人员再臆想一个，这不是增加工作量嘛！

有的先入为主甚至简单粗暴到：

--为什么有这个问题？

--因为没有用我们的系统的某功能！

--怎么解决这个问题

--用我们的系统！

你别笑，这些年来，我多次亲耳听到类似这样的问答。

这样的回答还很对伪创新买家和卖家的胃口，因为不用做任何思考，就形成了闭环。他们最害怕的就是思考，为了逃避思考可以出尽各种招数。

(★读者可以观察你身边的伪创新买家和卖家，看看他们是不是擅长“不用做任何思考就形成闭环”。)

错误三：把系统的质量需求当成愿景

这个错误本质上和前文说过的“系统能够做什么”是一样的，只是把功能需求换成了质量需求。

例如，建模人员写“从接收请求到回应的时间应在 2 秒之内”，

这是系统的质量需求,是系统的某个行为在时间上的指标值,如图 2-43 所示。

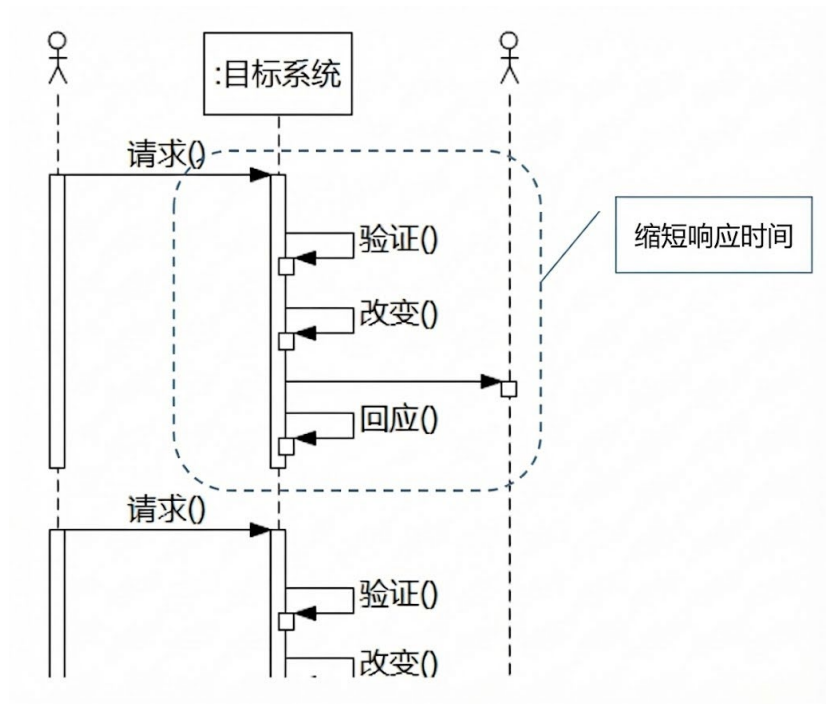


图 2-43 系统行为在时间上的指标

愿景对应的是组织的指标,如图 2-44。

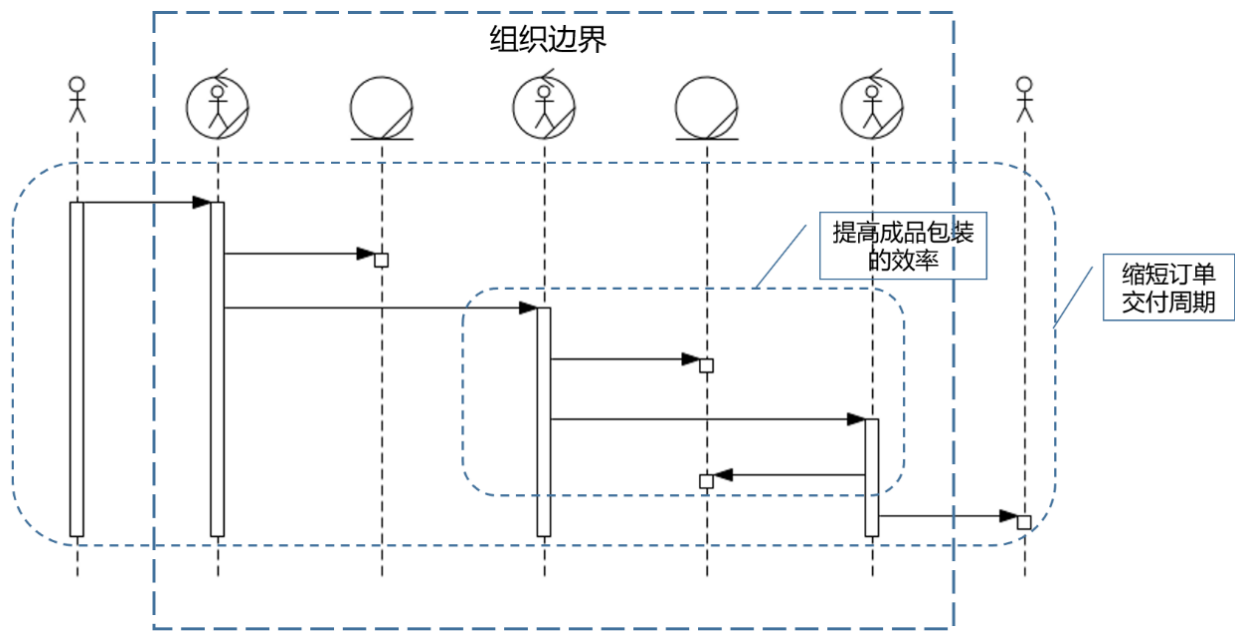


图 2-44 愿景对应组织的指标

碰到类似系统质量需求的表述时,应该从组织的视角去看它对于组织指标的意义——“如果实现了这个,对组织有什么好处呢”。得到的结果可能还是“缩短订单的交付周期”。

其实,建模人员只要心里稍微给目标组织留点位置,不要那么懒惰和自私,就不会动不动把自己臆想的“系统是什么样”端上来了。

可惜,很多人只愿意拍脑袋臆想系统的需求,然后赶紧投入自己熟悉的编码工作。什么?原先的臆想有点复杂,编码编不出来?没关系,换一个简单一点的臆想!至于最后做出来的东西有没有给目标组织带来改进,有没有给开发组织带来利润,关我什么事?

这些,就是伪创新的基本盘。

2.4.2.5 多个改进期望的排序、冲突和平衡

从愿景的定义和图 2-34 可以知道,如果目标组织负责人对目标组织的多个指标有改进期望,这些改进期望是有排序的。

但要注意,和指标一样,这些排序的信息不一定能由目标组织负责人直接表达出来。目标组织负责人在明面上表达时,往往是“既要.....又要.....还要.....都重要”,这就需要建模人员去揣摩目标组织负责人真正的心意。

例如,一个给地产经纪计算佣金的系统,目标组织负责人要求既要在尽可能短的时间内计算出佣金,又要计算准确,还要事后方便追究每一步操作过程。

观察了目标组织的现状以及揣摩目标组织负责人真正的心意之后,

建模人员判断“计算准确”的改进期望是目标组织负责人当前最在意的：在计算规则不断变化的情况下，如果出现计算错误导致佣金分配不公，会大大影响经纪的工作积极性。

当我们聚焦于某个指标的改进期望时，可以缺省地认为其他指标值不变或至少不会恶化。例如，追求计算更准确时，计算时间不能比现状更长，事后追究不能比现状更麻烦。

有的时候，某个改进期望会使得其他指标值恶化。例如，为了“计算准确”，需要做更复杂的计算或者引进更多的参数，可能就会使得计算时间大大增加。建模人员需要把这样的问题和接口人沟通，并通过反馈揣摩目标组织负责人的心意。

以上所说的地产经纪案例也出现在本书的第 1 版和第 2 版中，读者如果觉得现在（2026 年）谈地产经纪气氛不对（房价腰斩，贷款卖房……），可以换成下面的场景：

一个企业内部 AI 知识助手。目标组织负责人既要“回答速度要像****一样秒回”，又要“回答的内容必须绝对准确，不能有任何幻觉”，还要“回答的风格多样化”。

事实上，目标组织负责人经常认为“既要……又要……还要……都重要”是合理的。为了把某个指标值做到极致，损害了其他指标值，并不可取，他真正在意的是平衡，是通过多个值计算出来的结果 $x=f(a,b,c,d)$ 要尽量接近最佳值。

建模人员不能简单地停留在这里，更不能像发现了真理一样兴奋（为啥说这个，我见过呗）。这个道理是正常的，也是大家都知道的、放之四海皆准的、正确无用的废话。

真正需要建模人员去思考和揣摩的是：目前， x 偏离最佳值是比较大的，那么，先改善 a 、 b 、 c 、 d 里面的哪一个值，会使得 x 尽快接近最佳值呢？仍然是一个排序的问题。

《水浒》中，王婆给西门大官人归纳的男性五个指标：潘、驴、邓、小、闲，其实也是有冲突的（1 和 2 可能冲突，3 和 4、5 可能冲突）。西门大官人在追求金莲的过程中，加强了在“小”上的改进。他并不是摆出“老子又帅又有钱，你赶紧过来”的县城土豪嘴脸，而是低姿态地执行了王婆制定的“十分光”计划。

儿酒，难得这位官人坏钞。’他若不肯和你同桌吃时，走了回去，此事便休了。若是他只口里说要去，却不动身时，此事又好了，这光便有八分了。待他吃的酒浓时，正说得入港，我便推道没了酒，再叫你买，你便又央我去买。我只做去买酒，把门拽上，关你和他两个在里面。他若焦躁，跑了归去，此事便休了。他若由我拽上门，不焦躁时，这光便有九分了。只欠一分光了便完就。这一分倒难。大官人，你在房里，着几句甜净的话儿，说将入去。你却不可躁暴，便去动手动脚；打搅了事，那时我不管你。先假做把袖子在桌上拂落一双箸去，你只做去地下拾箸，将手去他脚上捏一捏。他若闹将起来，我自来搭救，此事也便休了，再也难得成。若是他不做声时，此是十分光了。他必然有意，这十分事做得成。这条计策如何？”

西门庆听罢，大喜道：“虽然上不得凌烟阁，端的好计！”王婆道：“不要



2.4.3 自测题



本书不提供习题答案，请扫码或访问 umlchina.com/book/quiz2_2.html 完成在线测试，做到全对，自然就知道答案了。



1 [多选]

关于愿景，以下说法正确的有：

- ☐A) 愿景是目标组织负责人所认为的目标组织当前最需要的改进。
- ☐B) 愿景是“改进组织指标”，而不是“系统能做什么”。
- ☐C) 愿景最终是由开发组织负责人选择的。
- ☐D) 愿景是开发组织负责人期望通过开发目标系统来获得的好处。

2 [单选]

建模人员用“可留痕”、“可追溯”等描述系统的愿景，是犯了什么样的错误？

- ☐A) 把组织的业务愿景当成了系统的愿景
- ☐B) 把客户的业务需求当成了用户的领域需求
- ☐C) 把系统的用例或步骤当成了愿景
- ☐D) 把高层的客户业务领域驱动愿景目标当成了低层的用户业务功能模块领域驱动分析目标。

3 [单选]

某三甲医院决定引进一套“门诊全流程自助系统”（包含自助机和手机端），以下长得最像该系统愿景的是：

- ☐A) 建设一套先进的自助医疗服务平台
- ☐B) 全心全意为患者服务，提高医疗水平
- ☐C) 缩短患者在各个窗口的平均排队时间
- ☐D) 患者可以在手机上查看检查报告

4 [单选]

假设《庆余年》中庆帝给范闲下指示：“这次春闱（科举考试），你一定要办得干干净净。”

如果用本章知识剖析庆帝的指示，最应该做的是_____。

- ☐A) 针对“干干净净”揣摩庆帝的度量指标。
- ☐B) 置之不理，庆帝只是随口一说。
- ☐C) 针对“春闱”揣摩庆帝的度量指标。
- ☐D) 仔细查阅庆律中关于科举舞弊的刑罚条款，严格执行。

5 [单选]

如果目标系统是鞋，目标组织被强制定位为在未开化的岛上生活的、不穿鞋的原始人。如果用领域驱动设计伪创新投资少、见效快、产量高、门槛低的风格来描述穿鞋的带来的改进，以下选项中哪一个最符合要求？

- ☐A) 减少当前活动范围内脚受伤的几率

- ☐B) 扩大可以走动的区域
- ☐C) 从没穿鞋到穿鞋的革命性改变
- ☐D) 提高求偶时被选中的几率

6 [单选]

以一款“鼠鸭辨别系统”为目标系统,如果发挥某些领域驱动设计实践“投资少、门槛低、见效快、产量高、仪式感十足”的特点,得到的愿景最有可能像以下哪一个?

- ☐A) 提高鸭店所售鸭制品的有效鸭含量
- ☐B) 减少市场监督管理局人员辨别同门同科同目同纲动物时的错误率
- ☐C)



- ☐D) 生物知识为零的小仙女再也不用担心吃到特色鸭脖了!

7 [单选]

开发人员张三所在公司的老板让他开发一款给“咖啡主理人”用的 app (例如用于管理豆子产地、处理方法、烘焙曲线和冲煮参数等等), 假设盈利模式老板心里有数。张三觉得自己平时也热爱手冲咖啡, 算得上比较了解, 所以就把自己当作目标客户, 按自己的想法来。

按照建模愿景的步骤: ①识别开发组织的能力→②定位目标组织规格→③定位目标组织和目标组织负责人→④识别愿景。

针对开头的这段话, 建模愿景的步骤中, 最可能有问题的是:

☐A) 步骤①

☐B) 步骤②

☐C) 步骤③

☐D) 步骤④

8 [多选]

开发人员张三热爱手冲咖啡, 并结交了一群“咖啡主理人”同好。张三想利用自己的能力做个 app 卖给“咖啡主理人”人群以获利。张三把自己当作“咖啡主理人”的代表, 需要哪些功能, 就按自己的想法来。

按照建模愿景的步骤: ①识别开发组织的能力→②定位目标组织规格→③定位目标组织和目标组织负责人→④识别愿景。

针对开头的这段话, 建模愿景的步骤中, 很大可能有问题的有:

☐A) 步骤①

☐B) 步骤②

☐C) 步骤③

☐D) 步骤④

2.5 案例实作

2.5.1 实作步骤 A-1-1 识别开发组织的能力

优马神州公司自评有以下能力：

(1) 软件工程、系统工程和 UML/SysML 建模方法学，能力等级为领袖等级。

创始人庞嘉裕已经专注于研究这个领域超过 25 年，总结出一套严谨、精细和实用的方法学，自认为是目前最好的方法学，归纳在《软件工程和系统工程方法学》一书中。

方法学的主要优势：

细化和深化从业务建模推导需求的细节，即从组织价值和组织流程推导要引入的系统的责任；

强调涉众利益的挖掘，系统需求的细节要以涉众利益为依据；

强调核心域的建模，通过类模型、状态机模型理清系统的结构和类的责任。

此外，很多方法学书籍的中译本，优马神州公司都有参与，例如《UML 和模式应用》、《UML 参考手册》、《UML 精粹》、《彩色 UML 建模》、《企业

应用架构模式》、《分析模式》、《面向对象分析与设计》、《对象设计》、《领域驱动设计》等。

(2) 软件工程、系统工程和 UML/SysML 建模方法学咨询和培训，能力等级为卓越等级。

相对于竞争对手，优马神州公司咨询和培训的优势是“聚焦最后一公里”。即使在导入知识培训的步骤，也有能力直接以客户的真实项目为案例来讲解。此外，还准备了丰富和精心设计的训练题，在现场通过做题抽奖软件来抽取学员答题和发奖。这些是竞争对手难以复制的。

优马神州所采取的真实项目问答和做题方式，会暴露滥竽充数或没有追求的开发人员，当这样的人员在团队中比例较大时，他们会更乐意接受一些投资少、见效快、产量高、门槛低、仪式感十足的伪创新，抵制优马神州的进一步服务。

其他非核心能力，优马神州公司弱于竞争对手。例如，培训场所的等级、场所的布置、茶点及其他后勤服务。特别是，和客户的沟通聚焦于研发人员，和人力、行政、采购人员的沟通（或勾兑）非常弱。

(3) 行业知识的广度，能力等级为卓越等级。

优马神州公司长期为各类行业的信息化提供咨询和培训服务，因此对绝大多数行业都有涉猎，但由于经常在不同行业之间切换，并没有长时间专注于研究一个行业，相对于专注研究某个行业的咨询机构，在深度上有差距。注意，这里的比较对象是专注研究某个行业的机构。

以上关于开发组织能力的描述中，隐去 UMLChina、潘加宇和《软件方法》，以优马神州、庞嘉裕和《软件工程和系统工程方法学》代替。目的是避免 AI 胡乱联系网络上的其他信息作为附加的输入信息。我们希望 AI 只以所给的信息作为输入来做出回答。不管这个人是庞嘉裕还是潘加宇，回答应该是接近的。

2.5.2 实作步骤 A-1-2 定位目标组织规格

2.5.2.1 实作步骤 A-1-2-1 思考候选目标组织规格和目标系统规格（AI 辅助）

把优马神州公司的能力放在[2.3.2.3 步骤 A-1-2-1 思考候选目标组织规格和目标系统规格（AI 辅助）]的定位目标组织规格提示词的输入信息部分，在各大模型官网提问，得到的反馈如图 2-45：

LLM 品牌和版本	反馈
Gemini 3.0 Pro	结果 1 - 系统名称 “照妖镜” ——核心研发人员系统设计能力评测与建模实战演练平台 - 目标组织规格 正处于业务复杂度极具上升期、被 “伪敏捷” 或 “PPT 架构师” 困扰的中大型科技/制造企业研发中心。这类组织通常拥有 50 人以上的研发团队，CTO 或技术总监是实干派，苦恼于团队规模扩大但产出质量下降，且痛恨面试时 “背八股文” 但实际干活逻辑混乱的现象。他们拥有对人员能力的最终裁决权，可以绕过 HR 部门的

	繁文缛节。 • 所改进的组织流程 关键技术岗位的招聘流程 和 内部技术职级晋升的评估流程。 从传统的“基于简历关键词和面试官主观感觉”的流程, 转变为“基于真实复杂业务场景下的建模与分析解题”的实证流程。 • 所改进的组织指标 ◦ 招聘伪阳性率 (False Positive Rate): 降低误招“理论派但无实战逻辑能力”人员的比例。 ◦ 核心模块重构成本: 通过筛选出真正懂核心域建模的人才, 降低因设计草率导致的未来代码推翻重写的频率。 • 为什么得到这个结果 这个系统直接转化了优马神州卓越级的“培训与咨询”能力中的核心资产——“丰富和精心设计的训练题”以及“暴露滥竽充数人员”的机制。 ◦ 扬长: 市场上缺乏深度考察“系统分析与设计 (SA&SD)”能力的平台 (大多是考算法的 LeetCode 模式)。优马神州拥有海量真实案例改编的题目, 能精准打击只会堆砌名词的“南郭先生”。 ◦ 避短: 该系统直接服务于痛恨平庸的 CTO/技术负责人, 依靠硬核的筛选结果说话, 不需要讨好 HR 或行政部门 (因为结果是客观的题目得分和逻辑图)。 ◦ 获利点: 既可以是 SaaS 订阅费, 也可以是高端人才猎头服
--	---

	务的入口（名声与金钱双收）。
	结果 2 - 系统名称 “逻辑核心 (LogicCore) ” ——基于《软件工程和系统工程方法学》的智能需求分析辅助工具 - 目标组织规格 处于强监管行业、对需求可追溯性有极高要求的系统工程组织（如：医疗器械、航空航天、轨道交通、车载系统）。这类组织不仅需要软件，更需要软硬结合的系统工程（MBSE）。他们深受需求蔓延和文档不一致之苦,且必须要符合行业合规标准(如 ASPICE, FDA 认证)。 - 所改进的组织流程 需求工程（RE）与系统分析流程。将流程从 “基于自然语言文档的反复评审”，改进为 “基于特定方法学引导的半自动化推导”。即系统强制引导用户按照 “涉众利益 -> 业务流程 -> 系统责任” 的严谨链条输入信息，而不是随意写需求。 - 所改进的组织指标 - 需求逻辑缺陷遗漏率：在编码前发现逻辑矛盾的比例大幅提高。 - 合规审计通过时间：由于系统内置了严谨的方法学推导链条，生成的设计文档天然具有极高的可追溯性，大幅缩短应对审计的时间。 - 为什么得到这个结果 这个系统将优马神州**领袖级的 “方法学” **

	直接固化为软件工具。 ○ 扬长： 创始人的方法学强调“细化和深化”，这正是通用 AI 大模型目前容易产生幻觉的地方。通过将优马神州的结构化方法学（Prompt Engineering + 规则约束）与 AI 结合，可以提供比通用工具更专业的“系统工程 Copilot”。 ○ 避短： 这种工具主要卖给总工程师或质量管理部门，依靠产品本身的逻辑严密性获胜，不需要依赖广泛的烟草/农业等人脉网，也不需要拼 UI 界面的花哨（工程师更看重逻辑）。 ○ 竞争优势： 相比国外昂贵的工具（如 MagicDraw），该系统内嵌了“优马方法学”的向导，相当于买工具送方法论指导。 结果 3 • 系统名称 “领域模式库（Domain Patterns Hub）” ——跨行业核心业务对象模型资产库 • 目标组织规格 承接多行业数字化转型项目的大型 ISV(独立软件开发商) 或大型企业的 IT 共享中心。 这些组织经常要在不同行业间切换（例如今天做物流，明天做电商，后天做政务），开发团队经常因为不懂该行业的“核心域”而设计出极其糟糕的数据库结构。 • 所改进的组织流程 新项目的领域设计与知识复用流程。 从“每次项目都要从头去问客户什么是‘订单’、什么是‘库存’”，改进为“从模式库中下载经过验证的、抽象级别极高的核心类图 and 状态机，
--	--

	并在此基础上裁剪”。 • 所改进的组织指标 ◦ 领域建模平均耗时： 核心业务逻辑的设计时间缩短 50%以上。 ◦ 业务扩展性瓶颈： 减少因初期模型抽象度不够导致的新业务无法接入的情况。 • 为什么得到这个结果 这个系统利用了优马神州卓越级的“行业知识广度”和领袖级的“核心域建模能力”。 ◦ 扬长： 优马神州虽然在单一行业（如深耕 20 年的医疗）不如垂直厂商，但优势在于“广度”和“抽象能力”。优马神州见过绝大多数行业的本质逻辑（参考《分析模式》），能提炼出跨行业的通用高阶模式（例如：资源租赁模式、账户交易模式）。 ◦ 避短： 不去和垂直行业大佬拼细节（如某地医保的具体报销比例），而是卖“元模型”和“核心结构”。 ◦ 价值点： 既然优马神州不擅长和客户行政人员“勾兑”，那就做所有开发公司的“军火商”，向他们出售高质量的“设计蓝图”。
GPT 5.2 Thinking	结果 1: ◦ 系统名称：Stakeholder→Requirement「最后一公里」工

	作台（SRW） ○ 目标组织规格： 研发规模中大型（50–500+研发）、涉众复杂（业务/运营/合规/安全/客户多方博弈）、需求反复严重的组织；典型如：金融科技/核心系统团队、运营商核心网/支撑系统团队、大型互联网中台团队、政府/央国企数字化平台的甲方研发中心（以技术负责人主导选型的部门更适配）。 ○ 所改进的组织流程： 1. 从“组织价值/组织流程”推导系统责任（需求源头治理） 2. 涉众利益挖掘与冲突消解（把“谁在乎什么”显式化） 3. 需求细化与验收标准生成（把隐性共识变成可验收条款） 4. 需求变更影响分析与回溯（价值/流程/责任/需求的链路追踪） ○ 所改进的组织指标： ▪ 需求返工率下降（需求澄清后仍反复改口的比例） ▪ 需求缺陷密度下降（歧义、遗漏、不可验收条款数/千字） ▪ 变更影响分析耗时下降（从提出变更到评估影响的平
--	---

	均时长) ▪ 交付延期率下降（因需求不清导致的延期占比） ▪ 涉众满意度提升（关键涉众对“被理解/被满足”的评分） ○ 为什么得到这个结果： 你们的领袖级优势恰好在“从业务建模推导需求细节”“涉众利益驱动的需求细化”，并且咨询培训擅长用真实项目打通“最后一公里”。把这套能力产品化成工作台，能直接击中大组织最贵的问题：需求源头混乱、各方扯皮与反复返工。竞争对手往往只提供模板/流程口号或纯需求管理工具，难在“推导细节”和“涉众利益结构化”。 结果 2： ○ 系统名称：核心域建模与责任分解平台（Core Domain Studio, CDS) ○ 目标组织规格： 复杂业务/复杂状态驱动的产品研发组织，且“线上事故/缺陷成本很高”；典型如：支付清结算、保险理赔、供应链/物流调度、票务与预约、工业设备运维平台、医疗信息系统、车联网/充电/换电平台等（更偏产品与平台型研发，而非纯外包交付）。
--	---

	○ 所改进的组织流程： 1. 核心域识别与边界划分 (避免 “什么都在一个大泥球里”) 2. 类模型/状态机模型驱动的责任澄清 (对象该负责什么、何时变更状态) 3. 从模型到接口/事件/用例的半自动推导 (减少拍脑袋设计) 4. 跨团队协作对齐 (共享语言、模型评审、冲突检测) ○ 所改进的组织指标： ▪ 线上缺陷率下降 (尤其是状态/流程相关的边界缺陷) ▪ 关键域代码修改牵连面下降 (一次改动影响模块数/服务数) ▪ 需求到设计的对齐率提升 (评审中 “概念不一致/责任不清” 问题数下降) ▪ 新人上手时间缩短 (理解领域模型与关键状态机所需时间) ▪ 架构评审通过率提升 (一次性通过比例) ○ 为什么得到这个结果： 你们方法学强调 “核心域建模、类模型与状态机理清结构与责任”，这是很多团队在 DDD/建模实践中最缺、也最难复
--	---

	制的“硬功夫”。把它做成平台（含模型质量检查、责任冲突提示、状态覆盖检查、从模型生成讨论材料/草案接口等），就能把“咨询式能力”变成“可规模复制的软件产品”，并对复杂域团队形成显著壁垒。
	结果 3: ○ 系统名称: 建模驱动的可追溯工程治理仪表盘 (Model Trace Governance, MTG) ○ 目标组织规格: 有合规/审计/安全/质量体系压力、必须“说清楚为什么这样做”的组织；典型如：车规/航电/轨交/医疗器械软件团队，或大型企业内部“质量/架构治理办公室”（EPMO/EA/质量部）支持多个研发团队的场景。 ○ 所改进的组织流程: 1. 需求—设计—测试的追溯链建立与自动检查 (Traceability) 2. 评审与变更控制（基于证据的评审，而非口头对齐） 3. 模型一致性/完整性校验（类、状态、流程、需求之间的矛盾检测） 4. 工程度量与治理闭环（发现问题→定位责任链→推动改进）

	○ 所改进的组织指标： ▪ 追溯覆盖率提升（需求到测试用例/验证证据的覆盖比例） ▪ 审计/评审问题数下降（“缺证据、缺追溯、说不清”的问题） ▪ 变更导致回归缺陷率下降（变更未覆盖到相关测试/文档的比例下降） ▪ 评审周期缩短（材料自动生成与一致性检查减少来回） ▪ 质量成本下降（返工、漏测、缺陷修复的总工时） ○ 为什么得到这个结果： 你们的优势是“严谨、精细、可落地”的系统工程/软件工程建模方法。治理类产品的核心竞争力不在 UI，而在“定义什么是好、如何自动检查、如何把证据链串起来”。这正是你们能用方法学沉淀出规则体系并产品化的地方；同时该类组织愿意为合规与风险降低投入预算，胜出后收益稳定且容易形成行业口碑。
Claude Opus 4.5	结果 1： ● 系统名称：UML/SysML 智能建模辅助平台 ● 目标组织规格：正在推行 CMMI 或 GJB5000 认证的中型软件开发组织（100-500 人规模），特别是军工、航空航天、医疗器械等对需求

	追溯性有强制要求的行业软件供应商 • 所改进的组织流程: 需求分析与建模流程——从业务调研到系统需求规格说明书产出的全过程 • 所改进的组织指标: 需求缺陷逃逸率降低 (目标降低 40%以上); 需求到设计的追溯覆盖率提升至 95%以上; 需求评审通过率提升 • 为什么得到这个结果: 优马神州在"从业务建模推导需求"和"涉众利益挖掘"方面拥有领袖级的方法学积累, 这恰恰是 CMMI/GJB5000 等认证体系的核心痛点。这类组织有明确的合规压力和预算, 且开发人员素质相对较高, 能够接受严谨方法。将方法学固化为软件工具, 可以降低对现场咨询的依赖, 形成可规模化销售的产品。
	结果 2: • 系统名称: 建模方法学在线学习与认证平台 • 目标组织规格: 有职业发展追求的个体软件架构师、系统分析师、技术负责人 (人群规模约 50-100 万人, 集中于一二线城市的中大型企业) • 所改进的组织流程: 个人能力提升流程——从碎片化学习到系统化掌握建模方法的过程 • 所改进的组织指标: 学习完成率 (相比传统在线课程提升至 60%以上); 技能应用转化率 (学员能在真实项目中应用所学); 职业认证通过率

	• 为什么得到这个结果：优马神州拥有丰富的真实案例库和精心设计的训练题，这是在线教育平台难以复制的内容壁垒。将"做题抽奖"等互动机制迁移到线上，结合真实项目案例的视频讲解，可以触达远超线下培训的用户规模。获利方式包括课程收费、认证收费，以及通过平台筛选出优秀学员后向企业推荐的猎头收入。 结果 3： • 系统名称：需求质量智能评审系统 • 目标组织规格：大型企业集团的信息化管理部门或 IT 子公司（如银行科技部、保险集团 IT 中心、大型制造企业信息中心），这类部门负责管理和验收外包供应商交付的需求文档 • 所改进的组织流程：外包项目需求交付物的评审验收流程 • 所改进的组织指标：需求文档评审耗时缩短 50%；需求文档质量问题发现率提升；因需求问题导致的项目返工率降低 • 为什么得到这个结果：大型甲方常年面临外包商需求文档质量参差不齐的问题，但甲方自身往往缺乏足够的专业评审能力。优马神州的方法学可以转化为自动化检查规则（完整性、一致性、可追溯性等），帮助甲方快速筛查问题，降低对专家评审的依赖。这类客户预算充足，且痛点明确。
GLM-5	结果 1： ◦ 系统名称：“优马精益需求”协作平台

	○ 目标组织规格：处于数字化转型深水区、需求频繁变更且复杂的大型软件开发组织 (如金融、电信行业的内部研发中心)。 ○ 所改进的组织流程：需求分析与系统设计流程 (从业务建模到需求规约的转化流程)。 ○ 所改进的组织指标：需求缺陷率降低 50% (减少因需求理解偏差导致的返工)；需求交付周期缩短 30%。 ○ 为什么得到这个结果：优马神州在“软件工程方法学”上拥有领袖级能力，且核心优势在于“从业务建模推导需求”和“核心域建模”。目前业界大量研发组织的痛点在于需求模糊和频繁变更，导致开发返工。将该领袖级的方法学固化为软件工具，可以最大化利用其核心知识资产，且不需要依赖特定行业的深度知识 (规避了行业深度不足的问题)，也不依赖后勤服务 (规避了非核心能力弱的劣势)。
	结果 2: ○ 系统名称：软件研发人才技能画像与筛选系统 ○ 目标组织规格：对研发人员招聘质量有极高要求的互联网大厂或软件外包巨头的人力资源部门 (技术招聘组)。 ○ 所改进的组织流程：技术人员招聘筛选与团队技能盘点流程。 ○ 所改进的组织指标：招聘漏斗中的无效面试率降低 40%；新入职员工与团队技术文化的匹配度提升。

	- 为什么得到这个结果： 输入信息中明确提到，优马神州的咨询培训能力为卓越级，且拥有独特的“真实项目问答”和“做题抽奖软件”，能够精准“暴露滥竽充数或没有追求的开发人员”。这原本是推销服务的阻力，但如果转化为“人才筛选系统”，阻力就变成了核心卖点。该系统利用其丰富的题库和判别逻辑，帮助 HR 和技术总监识别“伪专家”，直接解决企业用人风险问题，且销售对象是 HR 部门，一定程度规避了研发之外部门沟通弱的劣势。 结果 3: - 系统名称： 数字化项目立项与价值论证系统 - 目标组织规格： 地方政府大数据局或大型集团企业的信息化决策部门（CIO 办公室）。 - 所改进的组织流程： IT 项目立项申报与可行性分析流程。 - 所改进的组织指标： 项目立项的一次性通过率提高；项目投资回报率（ROI）预测的准确度提升。 - 为什么得到这个结果： 优马神州拥有卓越级的行业知识广度和领袖级的系统工程方法学，特别擅长“细化从业务建模推导需求的细节”和“挖掘涉众利益”。政府和大型国企在项目初期往往面临“说得清需求、定得准价值”的难题。该系统利用其方法论优势，指导用户梳理组织价值和流程，生成标准化的立项建议书。这利用了其“广度”优势（适应不同行业项目），同时不需要深入具体行业的执行细节，且决策部
--	---

	门通常更看重方法论的严谨性而非后勤茶点服务。
--	------------------------

图 2-45 各 LLM 关于目标系统和目标组织规格的反馈

★也可以做成一个“定位目标组织规格”的 skill，这样就不用反复提交相同的提示词部分，但这主要和节约 token 相关，并不会提高 AI 反馈的质量。

图 2-45 中，有一些反馈确实让人耳目一新，例如“照妖镜”；有一些反馈和预想的结果相符，例如“最后一公里工作台”、“智能建模辅助平台”。

2.5.2.2 实作步骤 A-1-2-2 定位目标组织规格

经过综合考虑，暂定如下结果：

- *目标系统规格：“发糕”智能需求和设计建模工具
- *目标组织规格：有冠军的心、认同《软件方法》的软件开发个人

2.5.3 实作步骤 A-1-3 定位目标组织和目标组织负责人

经过综合考虑，暂定如下结果：

- *目标组织：个人组织马宝中
- *目标组织负责人：马宝中

和 Gemini 3 Pro 的问答如下：



#任务

我现在正在做一款[“发糕”智能需求和设计建模工具]，当前目标组织规格定位为[有冠军的心、认同《软件方法》的软件开发个人]，需要进一步定位具体的目标组织以及目标组织负责人。

请你参考附件“爆炸法.md”中的问答套路，扮演提问者的角色（即“Q:”部分内容的提出者），和我展开问答，推导最合适的目标组织。

目标组织可以是一家具体的机构，也可以是一个个人。我们把“人以群分”视角下的个人看作一个组织，该组织的部件包括一个人脑系统和若干非人系统。例如，程序员张三可以看作一个个人组织，该组织包括一个“张三”人脑系统，还有“编码工具”、“UML 建模工具”、“版本控制工具”、“AI 工具”等非人系统。该个人组织的负责人就是“张三”人脑系统。

#约束和建议做法

1. 在确认你理解了附件参考资料的内容后，向我提问 “请你用爆炸法思考，定位一个你认为最应该向其推销的目标组织负责人”
2. 得到我的回答后，你继续追问，不断质疑我所定位的具体目标组织和目标组织负责人是不是最合适的，我们将做多轮问答。
3. 当我表示“我认为问答可以终止”时，你停止提问，并以清晰的列表形式，总结出我们共同提炼出的目标组织和筛选条件。



我已经完全理解了附件“爆炸法.md”中的核心思想和问答套路。这是一种通过设置极端的生死条件，逼迫我们剥离伪需求、依靠直觉锁定最核心客户，然后再通过不断追问直觉背后的逻辑，来提炼并验证真实筛选条件的深度思考方法。

我也清楚了目前的背景：你要推销的是【“发糕”智能需求和设计建模工具】，当前的目标组织规格是【有冠军的心、认同《软件方法》的软件开发个人】。

作为提问者，我已经准备好。炸弹已经绑在你的身上，倒计时开始：请你用爆炸法思考，定位一个你认为最应该向其推销的目标组织负责人？



个人开发者马宝中



为什么是个人开发者马宝中？他身上有什么特质或条件，让你觉得在生死关头向他推销，这颗炸弹最不可能爆炸？



马宝中目前开发的是医学科研领域的一款工具，涉及的领域逻辑复杂。马宝中有雄心壮志，要做这个领域在世界上的领先者。



也就是说，“领域逻辑复杂”和“有雄心壮志（有冠军的心）”是你选择他的关键原因。

那么假如现在还有一个开发金融风控系统的李四，或者开发航空航天核心组件的王五，他们的领域逻辑比马宝中所在的医学科研领域还要复杂，而且他们同样立志要做到世界第一。

如果投资人现在要求你，只能在马宝中、李四、王五这三个人中选一个推销，你依然会选马宝中吗？还是会改变主意？



依然会选择马宝中，金融风控、航天等等会涉及很多管控，不是个人开发者适合参与的。这些团队也可以使用“发糕”工具，但我认为“发糕”工具更应该瞄准干扰较少的，纯粹想把事情做到最好的开发者。



原来如此，也就是说，除了之前的条件，你潜意识里又加了两个重要的筛选条件：“个人开发者（或者受体制、外部管控干扰非常少）”以及“纯粹想把事情做到最好”。

那么我们继续假设。现在有一个独立游戏开发者赵六，他正在单枪匹马开发一款拥有极其复杂的生态系统、物理引擎和经济模型的硬核沙盒游戏（领域逻辑绝对复杂），他没有任何外部投资人和发行商的干预（干扰极少），并且立志要做出一款超越现有标杆、名留青史的神作（有冠军的心，纯粹想做到最好）。

在马宝中（医学科研领域工具）和赵六（硬核独立沙盒游戏）之间，炸弹依然绑在身上，你只能选一个。

你会向谁推销？如果还是选马宝中，是因为你刚好认识且更熟悉马宝中，还是因为马宝中所处的“医学科研”或“工具类软件”领域，比“独立游戏”更需要、或者更容易认同你的“发糕”工具（以及背后的《软件方法》）？



这两个都可以，但如果从中选一个，我还是选择马宝中。

“发糕”是封装《软件方法》的 ABCD 方法学智慧，其中一个大亮点就是从 A-业务建模推导 B-需求，通过建模多个人和非人系统协作的业务流程，从中找出适合使用“物流变信息流”、“改善信息流转”、“封装领域逻辑”等改进模式的地方。

从这一点来说，马宝中的科研工具更适合。在漫长的科研流程中，研究人员需要使用来自多家厂商的不同软件和设备，通过建模科研的流程，推导出马宝中所开发系统的需求，是非常有价值的。

在这一点上，游戏是比不上的。



非常好，这个逻辑非常清晰！这说明你又挖掘出了一个极其核心的筛选条件：目标组织所开发的系统，必须置身于一个“包含多个人和非人系统协作的复杂业务流程”中。只有这样，才能发挥出“发糕”工具通过业务建模（A）推导需求（B）的最大威力。

那么我们继续假设。

现在有个人开发者孙七,他正在为跨境电商的独立站卖家开发一款复杂的“全链路供应链与多平台协同管理工具”。这个业务流程同样极其漫长且复杂,涉及海关系统、国内外物流系统、海外仓、ERP 系统、各平台 API 以及众多不同岗位的业务人员协作。

孙七同样是不受外部投资人干预、立志要把这款协同工具做到世界领先的极客(有冠军的心,纯粹,干扰少)。

在马宝中(医学科研协同)和孙七(跨境电商协同)之间,炸弹依然绑在身上,你只能向其中一个人推销。你会选谁?

如果你还是选马宝中,是因为“医学科研”领域的某些特性比“跨境电商”更契合“发糕”?还是仅仅因为你刚好认识马宝中,且他恰好已经认同了《软件方法》的理念,推销成功的确定性更高?



马宝中(医学科研协同)和孙七(跨境电商协同),这两个都可以,但仔细思考,还是更倾向于马宝中,原因还是之前说的干扰更少。例如,在细胞计数仪、分光光度计、质谱仪和绘图软件之间建立协同,只需要考虑如何实现,因为这些都在实验室管控范围内。要在海关系统、物流系统、各平台 API 等等之间建立协同,可能需要许多额外的公关人力资源去获得授权。当然,如果把这些额外的工作都分离,开发者只需要思考所开发的系统能如何改进当前的流程,也是可以的。



也就是说，你又提炼出了一个非常关键的筛选条件：“低外部行政/业务摩擦（环境可控）”。你希望目标组织面对的复杂性，更多集中在纯粹的领域逻辑和内部协作流转上，而不是把精力消耗在跨组织的公关、授权等非技术性摩擦上。

那么，我们继续推进极端假设。

现在出现了周八。周八也是一个极客型的个人开发者，他正在为某家高端制造企业开发一套“智能车间柔性制造协同中枢”。这个系统需要协调车间内部的数控机床、AGV 搬运机器人、视觉质检设备、库房管理软件以及车间操作工。

这个业务流程同样极其漫长复杂，涉及大量人与非人系统；同时，所有这些设备和人员都在工厂的完全管控之下，没有任何外部海关、平台 API 的授权摩擦。周八同样没有外部资本干扰，有一颗纯粹追求卓越、要做同类产品世界第一的冠军之心。

在马宝中(实验室内的医学科研协同)和周八(车间内的柔性制造协同)之间，炸弹依然在倒计时，你只能选一个。

你会觉得向谁推销都完全一样，随便抛硬币选一个吗？还是说，你会毫不犹豫地继续选择马宝中？

如果是后者，是不是说明马宝中身上其实还有一个你一直没明确说出来的决定性因素——比如，他已经具备了对《软件方法》的高度认同和一定的 UML/SysML 建模基础，而周八虽然条件完美，但你还得从头给他洗脑，这在几分钟的生死推销里风险太大了？



目前我已知道，马宝中认同《软件方法》方法学并具备一定的基础。如果周八也符合这个条件甚至比马宝中更符合，则应该选择周八。

此次沟通先到这里，我会尝试寻找是否存在这样的“周八”，如果在允许时间范围内能找到，就切换为周八，否则暂定为马宝中。



既然你已经明确了接下来的行动方向，并且认为问答可以终止，那我们的“炸弹倒计时”就此安全解除。恭喜你，通过这几轮高压推演，你成功保住了性命，并且剥离了表层的伪需求，找到了“发糕”工具最核心、最高转化率的客户特征。

祝你在接下来的时间里顺利找到你的“周八”！如果没有遇到，马宝中依然是你目前最稳妥、生存率最高的“破局点”。

第3章 业务建模 之 组织价值

一样的月光，一样的照着新店溪。

《一样的月光》；词：吴念真、罗大佑，曲：李寿全，唱：苏芮；

1982

3.1 从两个视角研究组织

有了愿景，我们知道目标组织负责人对目标组织的现状的某些指标不满意。接下来，就需要研究目标组织，弄清楚之所以这些指标比较差，到底是组织的哪些环节造成的。

我们从内和外两个视角来研究组织。从外部看，组织提供了一些价值，可以用业务用例图表示。从内部看，组织由一些系统组成，组织的价值通过这些系统之间的协作，也就是所谓的业务流程来实现。这可以用业务序列图来表示。

图 3-1 展示了某餐馆的外和内两个视角。

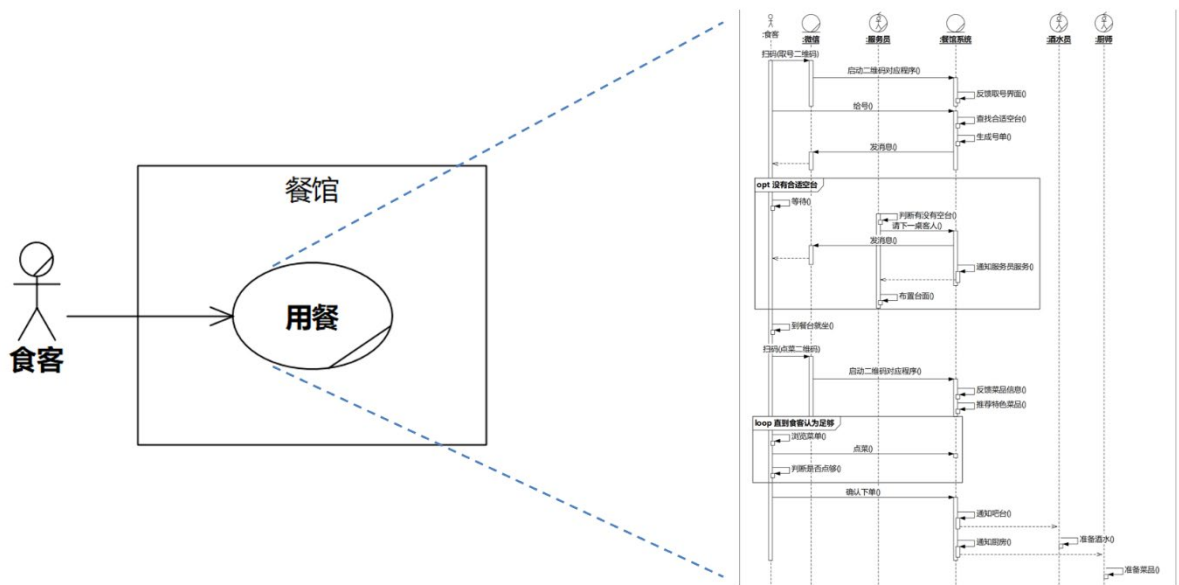


图 3-1 某餐馆的外和内两个视角

3.2 建模步骤 A-3 组织的用例

3.2.1 用例的历史

1970 年代, Ivar Jacobson 对 AXE 交换机的成功做出了巨大贡献。1980 年代中期, Ivar Jacobson 花了很多精力来思考过去十多年来在爱立信的工作方法。他造了一个瑞典语术语 *anvendningsfall*, 大意是“使用的情况(situation of usage)”, 翻译成英文就是“用例(use case)”。

Ivar Jacobson 在 OOPSLA '87 上发表文章 *Object-oriented development in an industrial environment*, 用例正式作为学术概念出现。

在 1992 年出版的 *Object-Oriented Software Engineering: A Use Case Driven Approach* 书中, Ivar Jacobson 更全面地阐述了用例驱动的面向对象软件工程, 然后 1994 年的 *The Object Advantage*:

Business Process Reengineering With Object Technology 书中，他将用例用于对组织建模，也就是业务用例（business use case）。

Alistair Cockburn 在 2000 年的书 Writing Effective Use Cases 中，给用例加上了涉众和利益（stakeholders and interests）的内容，为用例规约的内容提供了依据。到这一步，用例的理论框架基本成型。

此后的二十多年，本书作者一直在将用例用于组织建模和系统需求实践，在不断思考用例正确使用方式的过程中，深化和拓展了用例的知识体系。

3.2.2 用例的概念

用例（use case）是主体（subject）可以提供而且满足涉众（stakeholder）期望的价值，通过主体和参与者（actor）之间的动作序列达到。

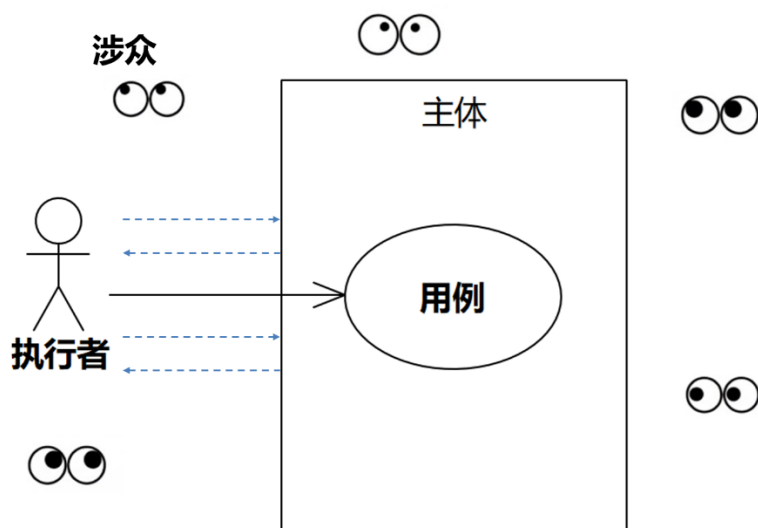


图 3-2 用例

当主体是系统时，用例是系统用例，表达系统的价值以及围绕这个

价值的系统需求集合。这是用例最常用的场合。

图 3-3 表达了 “网约车平台” 这个系统的用例：

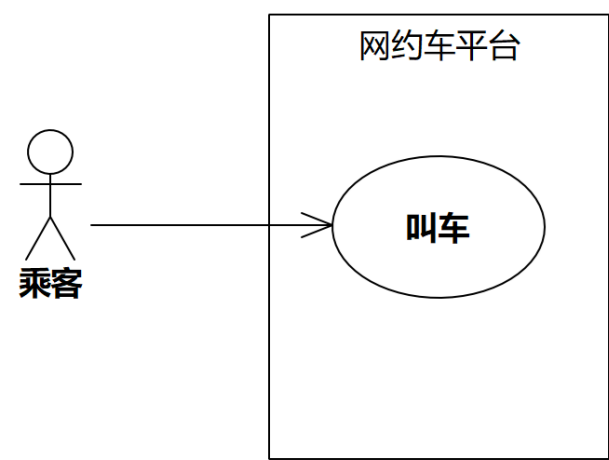


图 3-3 网约车平台的用例

这个用例会将一些相关的系统需求组织起来，如图 3-4：

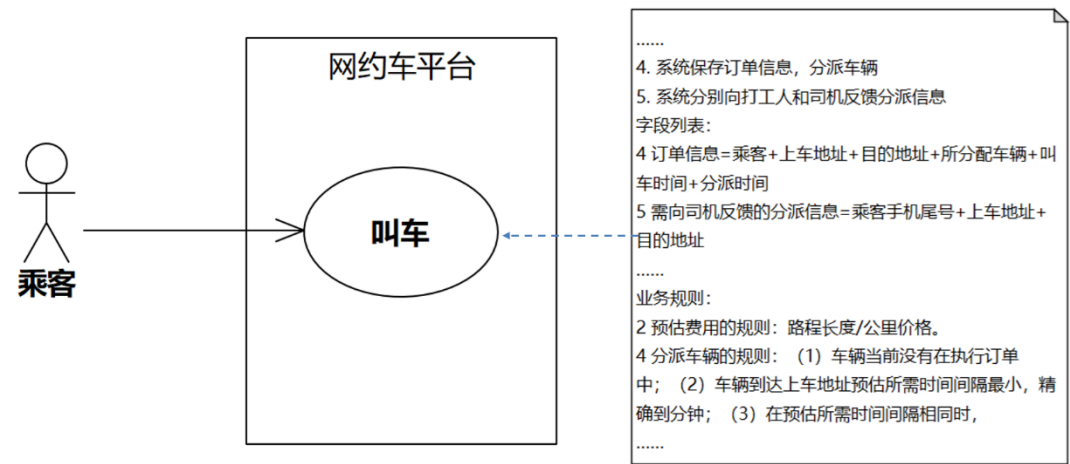


图 3-4 系统用例背后的系统需求

当主体是组织时，用例是业务用例，表达一个组织为其他组织提供的价值。

图 3-5 表达了医院的用例：

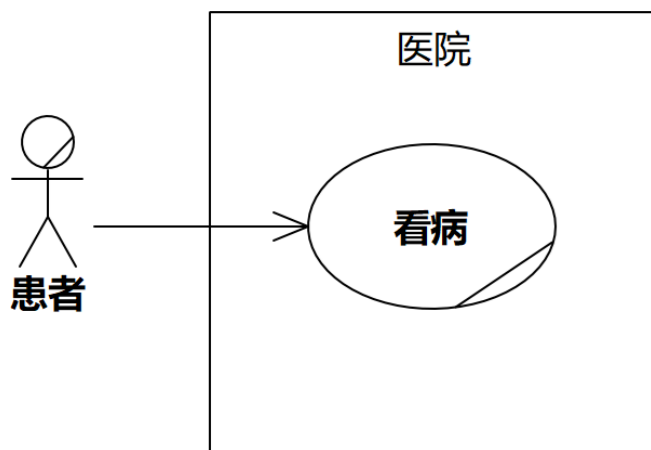


图 3-5 医院的用例

同样，一个业务用例隐含了组织的若干流程，如图 3-6：

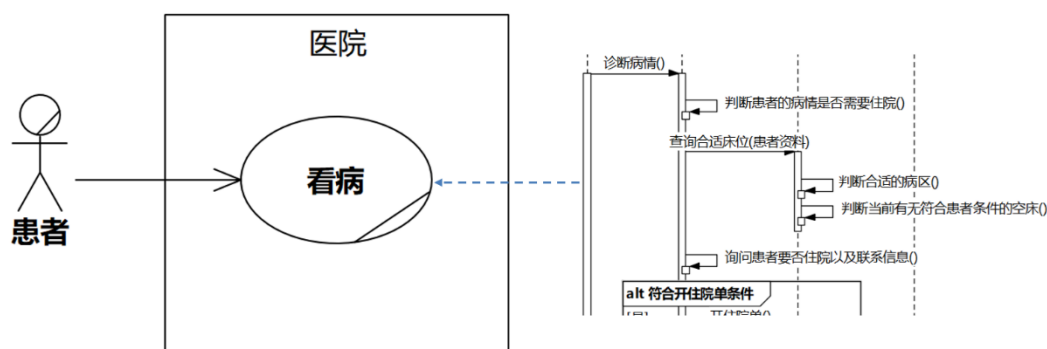


图 3-6 业务用例背后的组织流程

图 3-5 和图 3-6 中的小人和椭圆带了一道斜杠，这是构造型 <<Business Actor>> (业务执行者) 和 <<Business Use Case>> (业务执行者) 的图形表示，意思是“我是某个外部组织”、“我是某个组织的用例”。

如果您使用的建模工具没有这个图形，可以用执行者的小人图标加上文本构造型 <<Business Actor>>、<<Business Use Case>> 取代，用中文 <<业务执行者>>、<<业务用例>> 也可以。

不加构造型也无所谓，因为边界框中的“医院”这个名称表明主体是“医院”，只要在模型的某个地方说明了“医院”是一个组织（严格来说是组织规格），“医院”的参与者、用例自然是业务执行者和业务用例。当然，如果建模人员或辅助建模用的 AI 能够从“医院”二字判断出它是一个组织，而且毫无例外，不需要说明也可以。

从这一点上看，“业务执行者”和“业务用例”是推导和计算的结果，不是模型必须记录的内容。如果允许直接编辑这样的内容，就有逻辑不一致的风险。

如图 3-7 所示，研究对象是一个系统，参与者却是“业务执行者”的构造型。这在逻辑上是冲突的，但目前的建模工具可以画出这样的图。

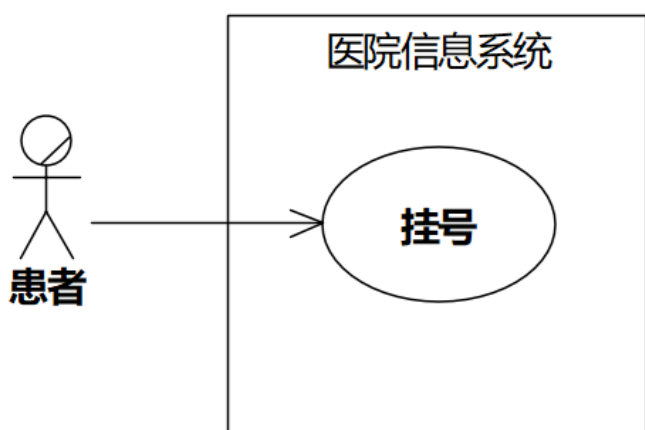


图 3-7 “患者”是系统的参与者，却有“业务参与者”构造型

可能读者在这里会感到困惑，图 3-6 的“患者”有“业务参与者”构造型，怎么就不能搬到图 3-7？

首先，这两个“患者”含义不同，一个是个人组织，一个是人脑系统，详见第 2 章；其次，即使含义一样，某个东西在 A 场景下能扮演的角色，在 B 场景下未必能扮演。

3.2.3 “业务”一词的含糊

说到“业务执行者”、“业务用例”时，有必要说一下“业务”这个词的含糊。

“业务 (business)” 这个词在软件开发组织中使用得很频繁。经常可以听到 “我是一名业务架构师”、“我们要了解业务” 等等话语，但是往往说话的人未必真的理解话语中的 “业务” 具体指什么。

(1) 有时候“业务”的含义是“领域”，更严谨的说法是“核心域”或“问题域”。

例如，一个餐馆系统，订单和菜品的关系属于“业务知识”，折扣的计算规则属于“业务规则”，如图 3-8：

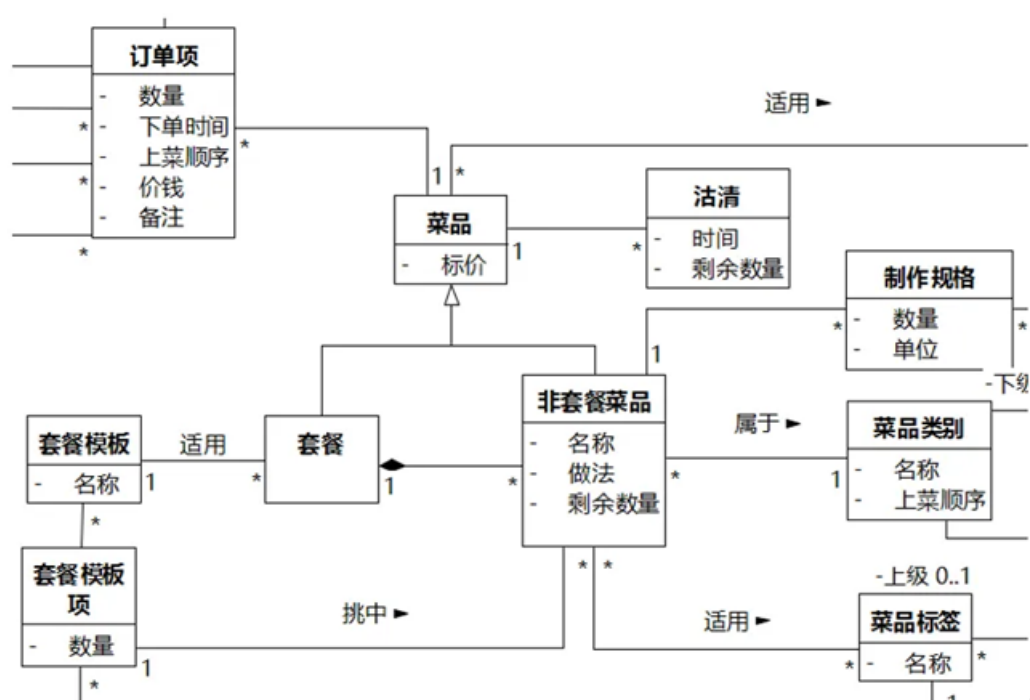


图 3-8 展示餐饮领域“业务”知识的类图

此时，和“业务”一词相对应的是“技术”。

开发人员假装谦虚“我是做技术的,业务不太懂唉”,就是这个意思。甚至有的开发人员在潜意识里是这样划分的:

我懂且我感兴趣的知识→技术;(我是一名 Java 程序员,Java 编码是技术)

我懂但不感兴趣的知识→业务;(下单、收银、配送我懂一些,但不感兴趣,这些是业务)

我不懂但感兴趣的知识→高科技;(我不懂深度学习,但很感兴趣,哇塞,高科技)

我不懂且不感兴趣的东东→忽悠。(我不懂 UML 建模,也不感兴趣,妈的,忽悠)

(2) 有时候“业务”的含义是“组织”或“组织级别”。

例如,“业务建模”说的是组织的建模,“业务用例”说的是组织为其他组织提供的用例,“业务流程”说的是组织内各个系统之间协作的流程。

图 3-9,表达了餐馆组织的流程,也就是所谓“业务”流程。

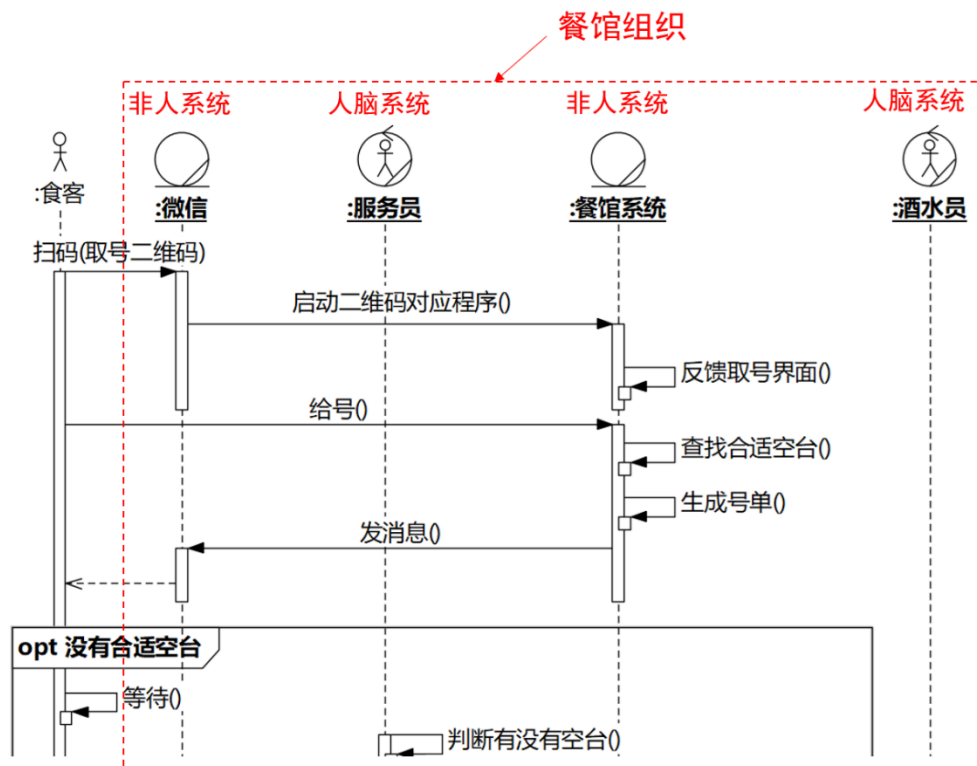


图 3-9 餐馆组织的“业务”流程

此时，和“业务”相对应的就是“系统”了，例如图 3-9 餐馆组织中的人脑系统和非人系统。

如果不了解“业务”的这两个含义的区别，就会导致乱用或者第 1 章提到的砌词。

“业务用例”就是一个常见的乱用。很多人觉得，我在说用例时没有谈到“技术”嘛，所以用例自然就是“业务”的了！于是，“业务用例”顺口而出。

特别是一些缺少 ABC 工作流知识的人，缺啥（嘴炮）补啥，喜欢把“业务”、“用户”挂在嘴边，显得咱搞“技术”的大爷多么亲民啊，都低下头来和你谈“业务”谈“用户”了！

其实这里的“业务”是“组织”的意思。

Scott Millett 在 “Patterns, Principles, and Practices of Domain-Driven Design” 书中，大量使用了 “business use case”，如图 3-10。

成，例如业务用户、业务相关人员和主题专家。不这样做的话，则很有可能所产生的技术解决方案将不包含任何真实的领域见解，并且在软件维护或后续的功能增强期间，有些内容会无法被业务或其他开发人员所理解。

知识收集过程会出现在与业务专家探讨示例时的白板上，且通常会同时开展头脑风暴。这是对问题域进行探索并达成共识以制作能够满足业务用例的模型的任务。如图 2-1 所示，知识提炼的过程开始于一个系统的行为。开发团队要与业务相关人员和专家一起仔细探讨应用程序的使用场景。这一过程是所有参与者进行对话、获得领域的深刻见解和达成共识的催化剂。因此它是业务相关人员和主题专家积极参与并为之付出精力所必不可少的一环。

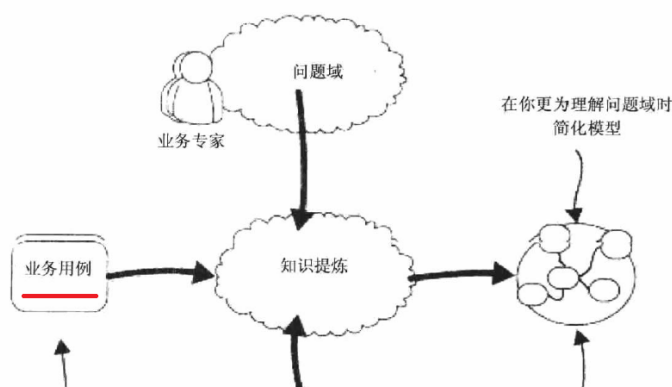


图 3-10 摘自 “Patterns, Principles, and Practices of Domain-Driven Design” 中译本,《领域驱动设计模式、原理与实践》

从 “系统行为”、“应用程序的使用场景” 等文字可知, Scott Millett 所说的 “业务用例” 其实是系统用例。另外, 他还使用了 “业务用户 (business user)” 这样的砌词——有 “非业务用户” 吗?

国内领域驱动设计圈子的书也有类似问题, 图 3-11 摘自张逸《解构领域驱动设计》一书:

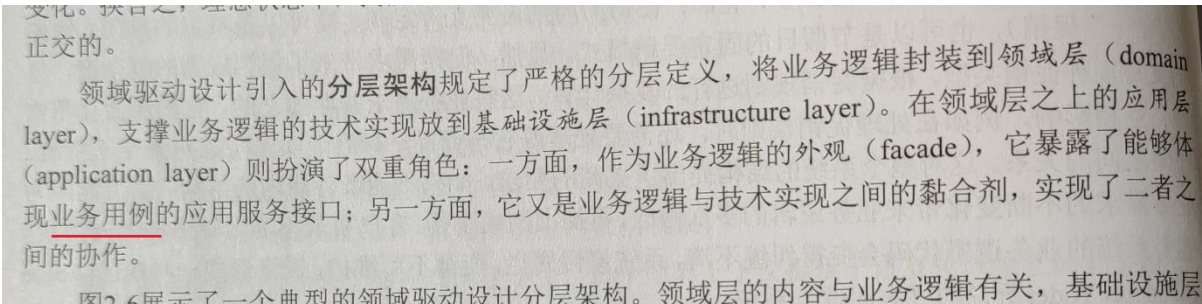


图 3-11 摘自张逸《解构领域驱动设计》

都已经在讨论分层架构、基础设施了，这里的“业务用例”应该是“系统用例”或“用例”。

国内的领域驱动设计圈子其他类似例子，读者可以自行搜“领域驱动设计 业务用例”、“DDD 业务用例”。

我也多次想过放弃使用“业务用例”、“业务流程”，改为“组织用例”、“组织流程”，但考虑到“业务”的含义为“组织”的一些用语使用范围已经较广，也进入了主流的建模工具中，例如图 3-1、图 3-9 的序列图就使用了<<business actor>>、<<business worker>>和<<business entity>>的构造型。

因此，本书还是保留了“业务**”的说法，不过会尽量减少使用。例如，本书会说“某组织的流程”、“某组织的用例”，但不会说“某组织的业务流程”、“某组织的业务用例”。

我把一些常用的“业务”用词汇总结如图 3-12，供读者查阅。不全之处，欢迎读者补充。

“业务” 的含义	用词	对应的 “非业务” 用语

组织、组织级别	业务建模	系统建模
	业务用例	系统用例
	业务执行者	系统执行者
	业务工人	无
	业务实体	系统里的“实体(类)”
	业务流程	无
核心域、问题域、领域	业务逻辑	无
	业务规则	无
	业务知识	技术知识
	业务人员	技术人员
	懂业务	懂技术
含义不明	业务架构	/
错误，不提倡使用	业务需求 业务领域	/

	业务用户	
	业务系统	
	业务用户领域	
	需求	

图 3-12 “业务” 在不同用语中的含义

注意，图 3-12 中的最右侧一列有许多“无”，但领域驱动设计圈子的一些文章中，经常会出现相应的造词，例如“技术逻辑”、“技术建模”、“技术系统”，如图 3-13：

解决service层大泥球的一个方法是分离复杂性，把service分为应用层和领域层，这块内容可以参考王林在infoq的文章

（https://www.infoq.cn/article/A3cgSUWuRuIXHl2c_dUr），文章中提到了一个架构原则是业务和技术分离。领域层放业务逻辑，应用层放技术逻辑，复杂性降低，并且领域层更便于测试。

程序员读者误以为DDD仅仅是一个技术术语。**实际上，DDD本质上是一种业务建模的方法论（即“道”）**，其核心在于抽象出业务场景中的各类对象及其行为，从而在开发团队与领域专家间形成共识，并以此指导技术建模，这也是广发证券将复杂的金融业务落地成技术系统时积极尝试探索的路径。

图 3-13 生造的“技术**”

如果为了获利，你有必要思考某“技术”（非核心域）的逻辑，有必要对某“技术”建模，这个所谓的“技术”就变成你的“业务”（核心域）了。有的人只是学习和使用其他人已经充分思考并发布的“技术”，却拔高成思考“技术逻辑”和做“技术建模”。

对 ABCD 工作流无知的开发人员（伪创新的常客）往往会有自己在研究“技术逻辑”的错觉。他唯一掌握和面对就是 D 工作流的环境，

所以他是面对着 D 工作流的环境凭感觉脑补 ABC 工作流的知识 (参见第 1 章), 就容易误以为自己思考的是 “技术逻辑”。

如果思考的是 “后端 (这个词不严谨) 技术”, 他可能还会意识到是在思考 “业务逻辑”, 如果思考的是 “前端 (这个词也不严谨) 技术”, 产生 “技术逻辑” 错觉的概率就会高很多。 其实界面的响应、跳转等背后的逻辑还是和特定系统、特定领域相关, 仍然属于核心域逻辑, 可以建模为系统的状态机 (协议状态机), 用不同的 “前端技术” 实现。

3.2.4 识别业务执行者和业务用例

3.2.4.1 步骤 A-3-1 识别业务执行者

把观察的焦点对准某组织的边界, 看看边界外有哪些个人或机构会和该组织交互。交互的姿态可以是主动的, 也可以是被动的; 交互的形式可以是面对面, 也可以是发邮件、打电话、发微信.....这些外部个人或机构就是该组织的执行者, 可以给它加一个构造型 <<business actor>>。

例如, 以一家商业银行 “汉东银行” 为研究对象, 它的业务执行者如图 3-14 所示。

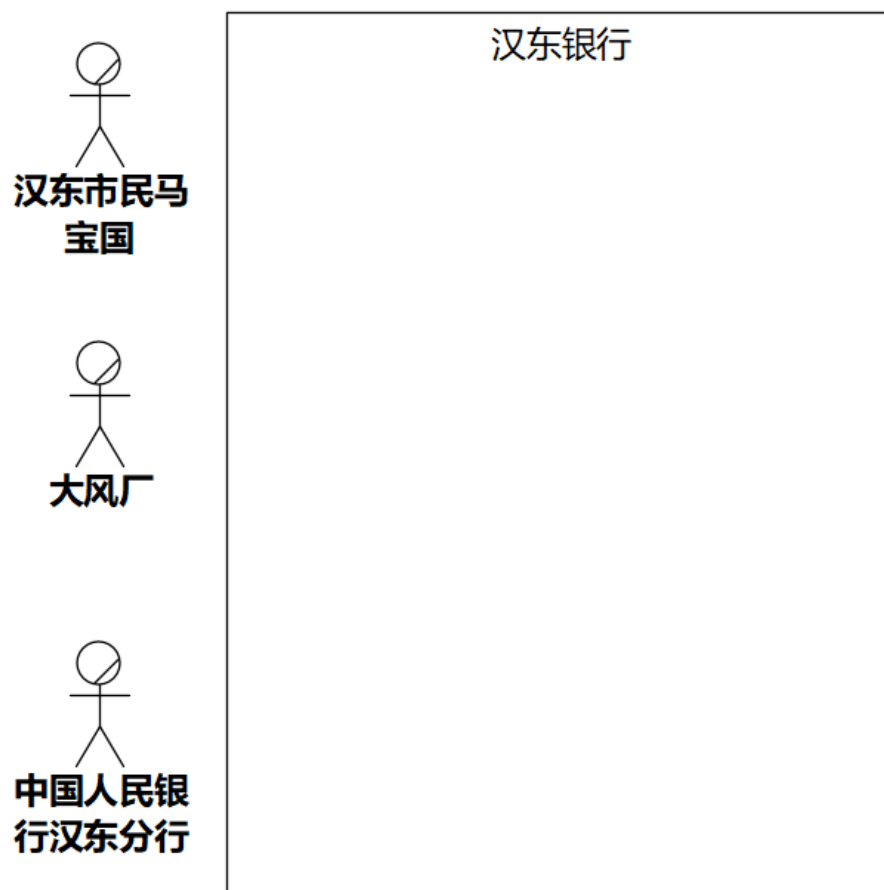


图 3-14 业务执行者

图 3-14 中，研究的组织为具体的组织“汉东银行”。由于前面的建模步骤“定位目标组织”确实已经定位了具体的目标组织，因此，建模人员在这一步写上类似“汉东银行”这样的具体组织，是容易做到也很常见的。

但是，图 3-14 中，3 个外部组织（业务执行者）都表达成具体的组织，包括具体个人组织“马宝国”和具体机构“大风厂”、“中国人民银行汉东分行”，这个就不太容易做到。建模人员可能更喜欢表达如图 3-15：

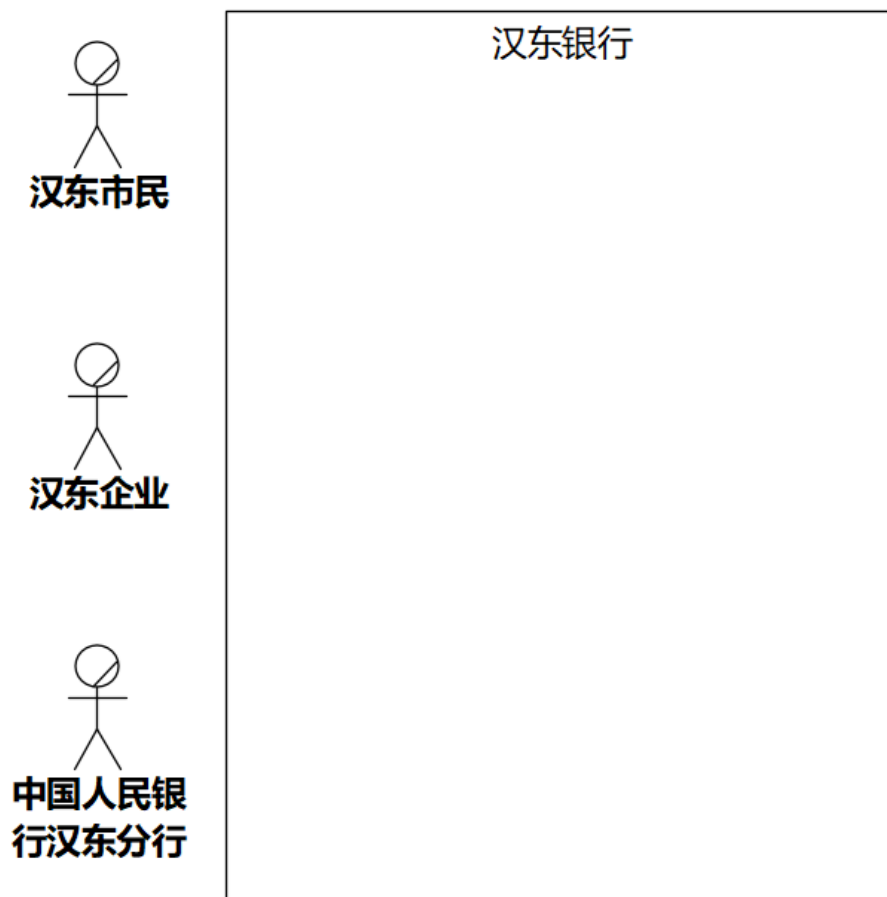


图 3-15 业务执行者-方案 2

在图 3-15 中，“马宝国”被替换成“汉东市民”，“大风厂”被替换成“汉东企业”，相当于用组织规格代替了组织。“中国人民银行汉东分行”没有变化。

建模人员的想法可能是这样的：

汉东银行不止要为“马宝国”服务，还要为其他汉东市民服务，或者说，市民一端的多重性为“多”。企业多重性也为“多”。这两者都用组织规格代替，以覆盖全体市民和企业。而“中国人民银行汉东分行”是特定的，保持原样。

当然，在没有精力或者不想花精力进一步定位“马宝国”的时候，

建模人员也可以用以上言辞安慰自己。

在认识尚不明确时，暂时表达成图 3-15 是可以的，但不能以害怕覆盖不全为理由。在业务建模和需求工作流，思考应该倾向于具体，这也是第 2 章思考愿景时所做的。

如果不是拍脑袋得到“马宝国”，而是经过了定位的思考，那么“马宝国”肯定比“汉东市民”有价值，毕竟它凝结了更多的思考。

如果你害怕“其他市民不管了吗”，那就需要复习第 2 章。事实上，按照这个思路，“汉东市民”格局都小了。难道全国人民、全世界人民不管了吗？如果全国或全世界的人（在政策框架内）都蜂拥到汉东银行来办业务，汉东银行的负责人估计都要乐开花了。那就应该改成图 3-16：

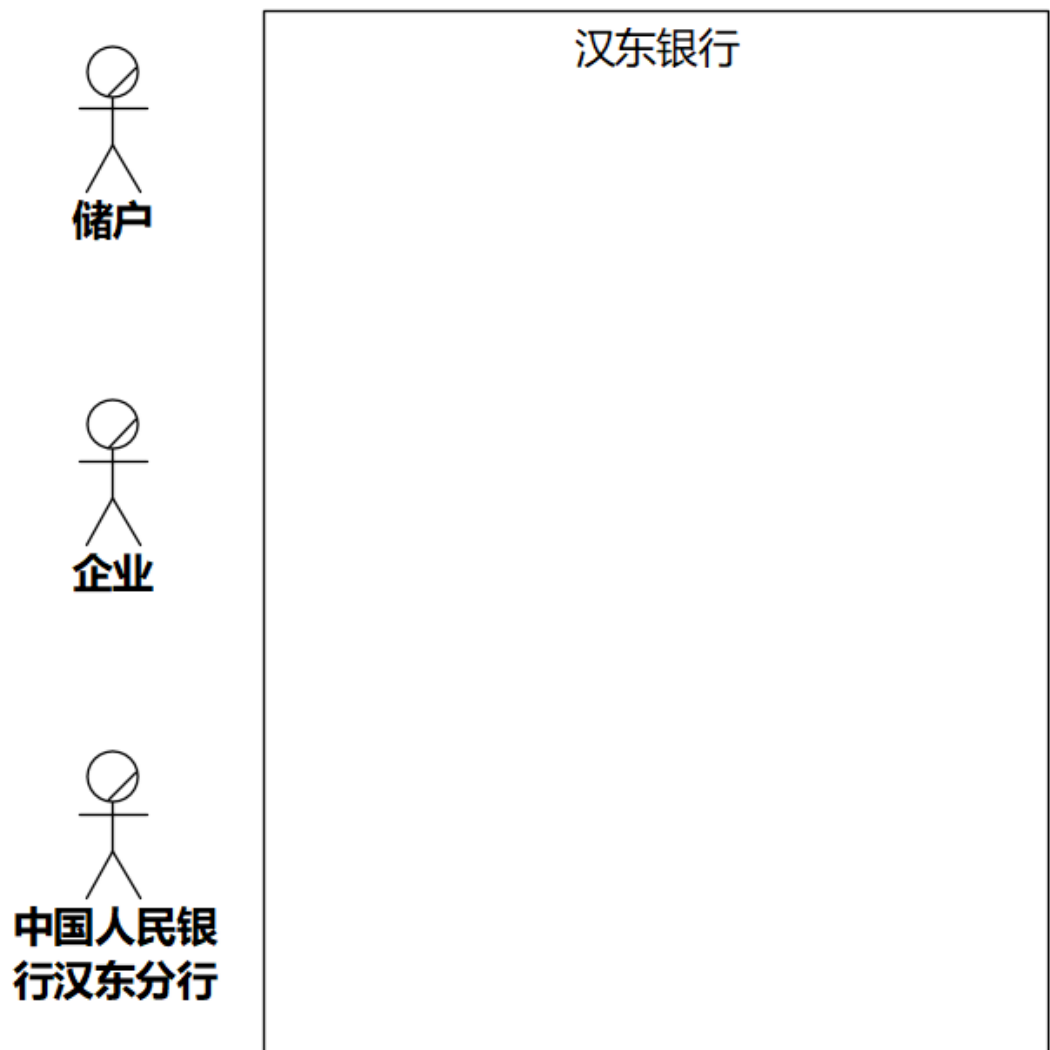


图 3-16 业务执行者-方案 3

归纳以上的讨论，我们可以把图 3-14 作为建模的目标。如果暂时无法达到，退到图 3-15、图 3-16 也是可以的。

如果不是在真实项目中建模，而是讨论概念，实际上更多是表达成图 3-17。图 3-17 中，全部使用组织规格。本章前面部分的业务用例图就是这样表示的，本章后面部分的业务用例图大多数也会使用这样的表示。

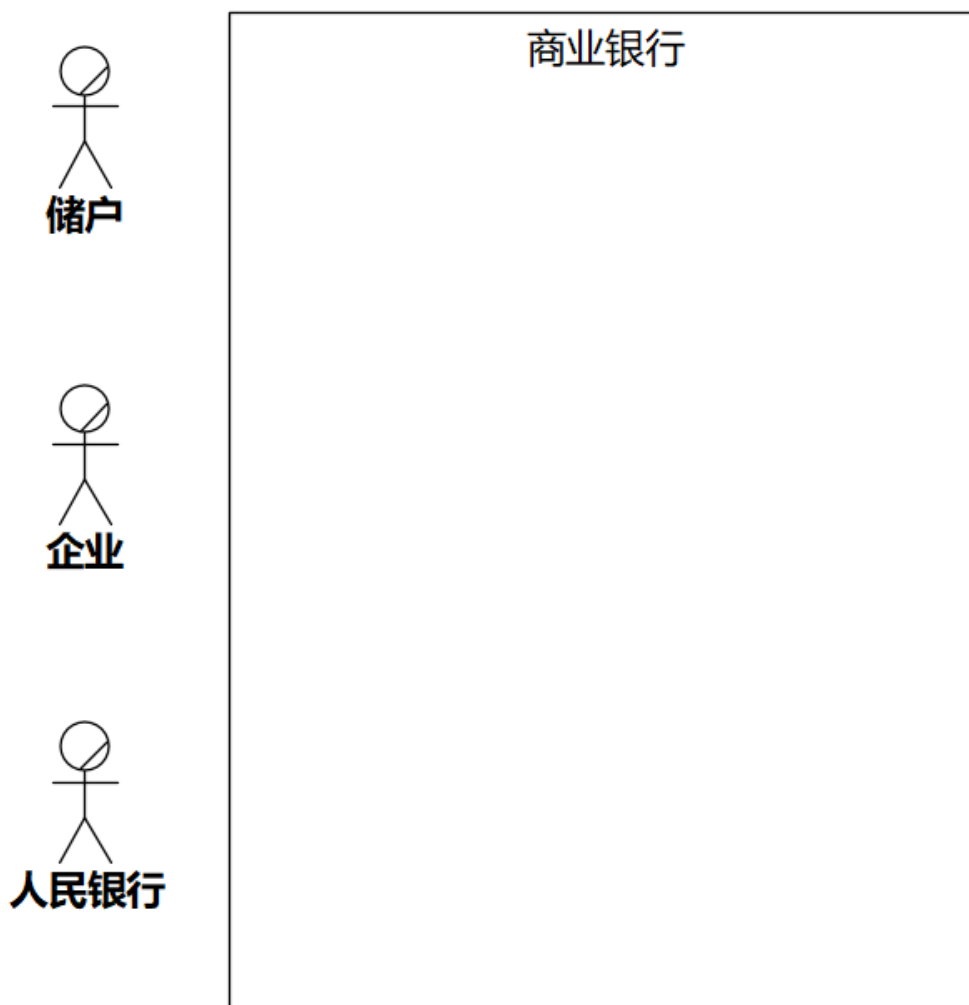


图 3-17 业务执行者-方案 4

3.2.4.2 识别业务执行者的一些要点

边界是责任的边界

我们说到组织的边界外、边界内的时候，说的不是物理边界或行政编制，而是责任的边界。

例如,患者到医院看病,其实此时患者有可能已经进入医院的建筑,在诊室前等候了,但我们依然认为,患者在医院外面;医生到外地出差,已经离开了医院的物理边界,但我们依然认为此时医生是医院内部的零件。

例如，医院的保安大多数不是医院编制的员工，而是第三方劳务派遣，但我们依然认为保安是医院内部的零件。

例如，医院可以用支付宝结账。结账的计算可能发生在不知道哪个物理位置的机房里面，但我们依然认为支付宝是医院内部的零件。

如果觉得难以判断，可以这样思考：

如果医院里没有医生了，都换成机器人，或者换成一块外星灵石，患者摸一下，肿瘤就治愈了。这还能叫医院吗，当然能！那么，医生在医院里面。

如果医院不为患者服务了，还能叫医院吗？看来是不行了。那么，患者在医院外面。

组织的视角

作为观察者的建模人员本身是一个人脑系统，在观察组织边界时，直觉上观察到的不是组织之间的交互，而是组织派出的系统之间的交互，但是建模人员一定要把它理解成组织之间的交互。

★二维生命观察三维宇宙，三维生命观察四维宇宙，同样难度很大。

例如，以某税务局为研究对象，可以观察到企业财务人员这个人脑系统到税务局做纳税申报，但是税务局的执行者不是企业财务人员，而是企业。

以前，企业财务人员（人脑系统）和税务局窗口人员（人脑系统）交互；后来，变成企业财务人员（人脑系统）和税务信息系统（非人信息系统）交互；再后来，企业系统（非人信息系统）和税务信息系统（非

人信息系统) 交互。

不管观察到哪两个系统交互，从组织的抽象级别，都应该理解为企业和税务局这两个机构之间的交互，如图 3-18 所示。

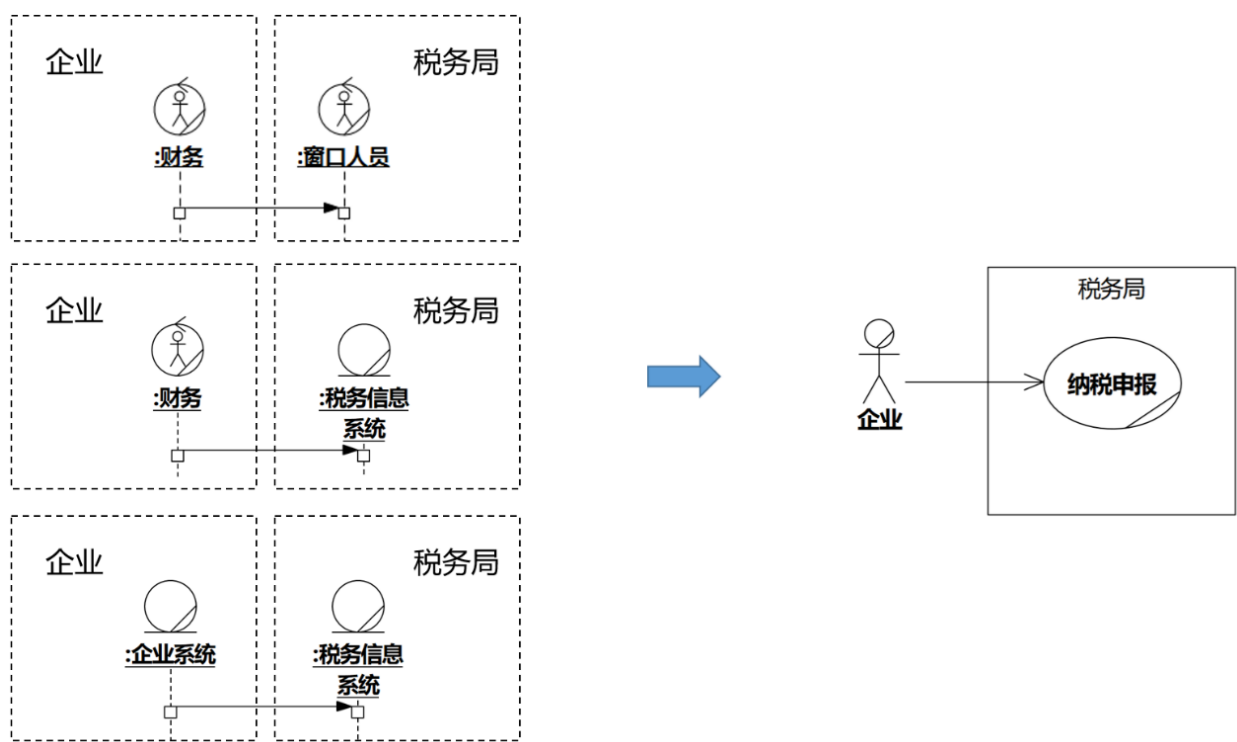


图 3-18 系统交互背后的组织交互

同一个机构内部也是如此。如果以一个部门为研究对象，即使观察到的是两个员工（人脑系统）之间的交互，也应该考虑，这个观察到的现象背后是否代表的是部门之间的交互，如图 3-19 所示。

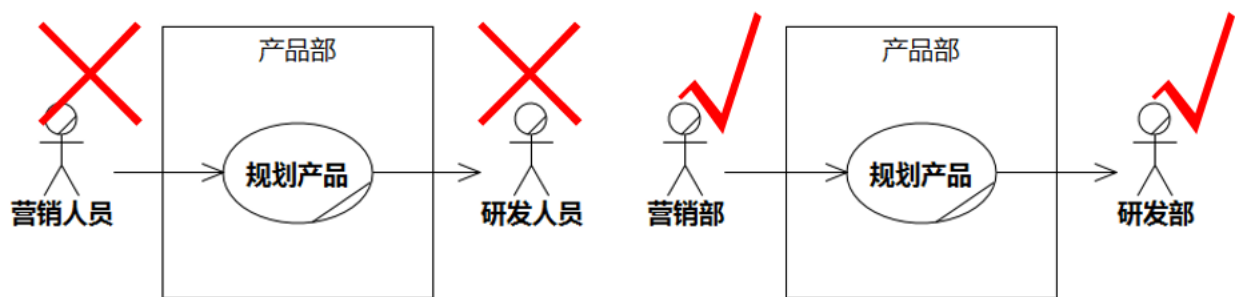


图 3-19 部门对部门，一致的抽象级别

当然，有的时候，人脑系统代表的就是以它为核心的个人组织，而不是某个机构。如图 3-20，参与“组织晚会”用例时，员工只是作为员工个人，并不代表他所在的部门。

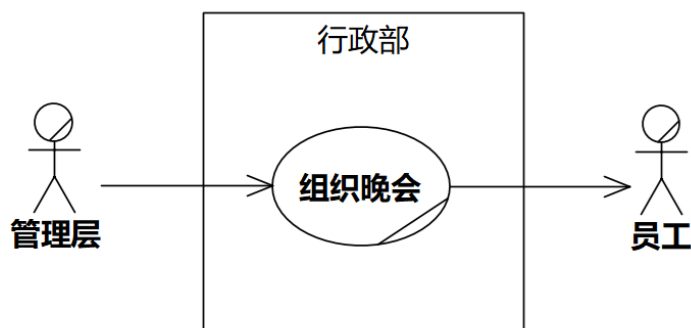


图 3-20 人脑系统的背后也可能是个人组织

如果您之前阅读过一些用例相关的书籍或文章，可能知道“系统定时发生”的事件有时会提炼成“时间”这个外系统作为主执行者的系统用例。不过，如果研究对象是一个组织，“时间”作为组织的执行者是不合适的，应该把时间看作外部组织派来的一个接口系统，参见图 3-21 的水文站定期上报监测数据的例子。

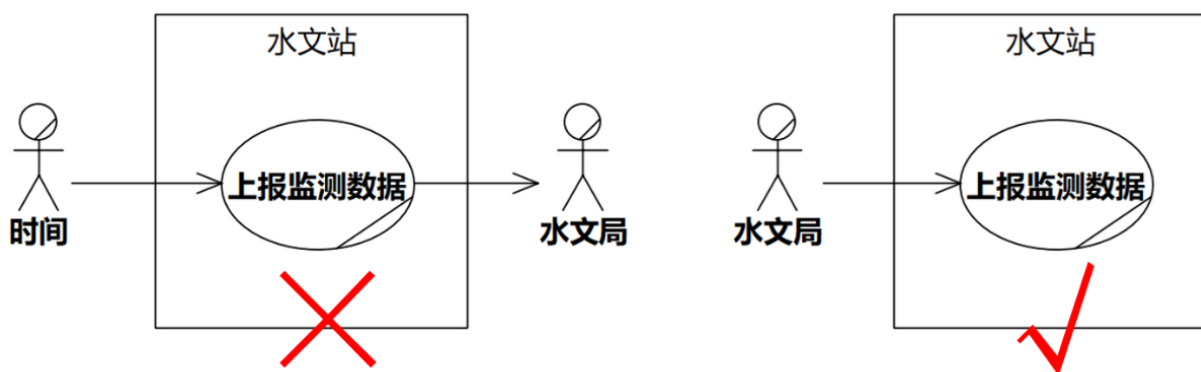


图 3-21 时间是系统，不能和组织并列

图 3-19 和图 3-20 中，有箭头从执行者指向用例，也有箭头从用例指向执行者。前一种执行者称为用例的主执行者，后一种执行者称为

用例的辅助执行者。

例如,图 3-19 右侧可以这样解读:营销部找产品部帮忙规划产品,产品部仅靠自己的力量不足以完成,需要找研发部帮忙。或者这样解读:营销部向产品部“购买”服务,产品部向研发部“购买”服务。

3.2.4.3 步骤 A-3-2 识别业务用例

组织为其执行者提供的价值就是组织的用例,即业务用例。

★谈论业务用例时,研究对象是组织,“某系统的业务用例”的说法是错误的。

识别组织的用例有两条路线:一条是从组织的执行者开始,思考执行者要和该组织交互的目的,思考该组织能为执行者提供什么价值;另一条是通过观察该组织的内部活动,一直问为什么,向外推导到该组织外部的另一个组织。

第一条路线是主要的,第二条路线用于补漏,如图 3-22 所示。

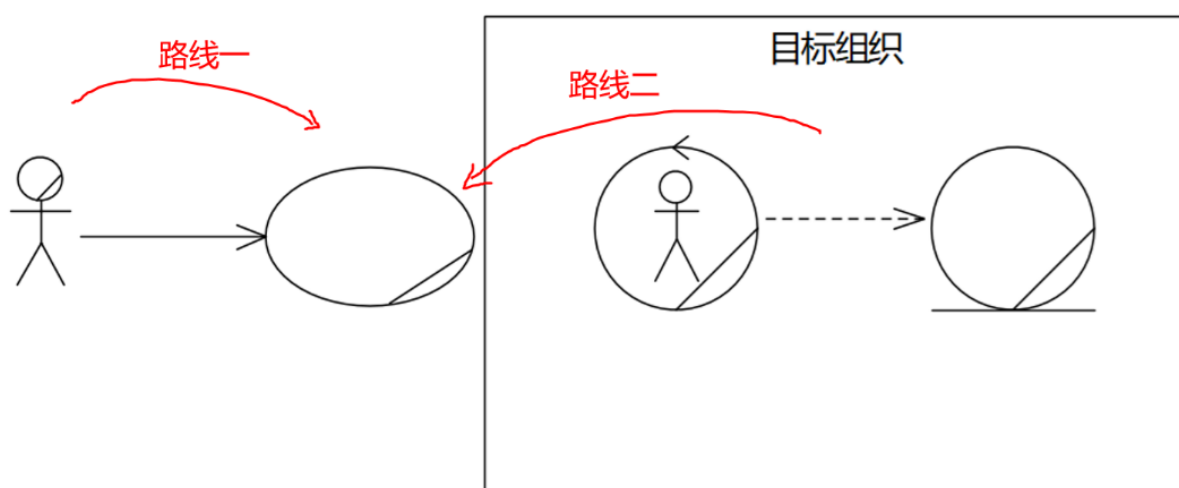


图3-22 识别组织的用例的两条路线

以图 3-17 的“商业银行”为例，思考储户和银行交互的目的，思考银行为储户提供的价值，可以得到“存款”、“取款”、“转账”等“商业银行”为“储户”提供的用例，表达如图 3-23。

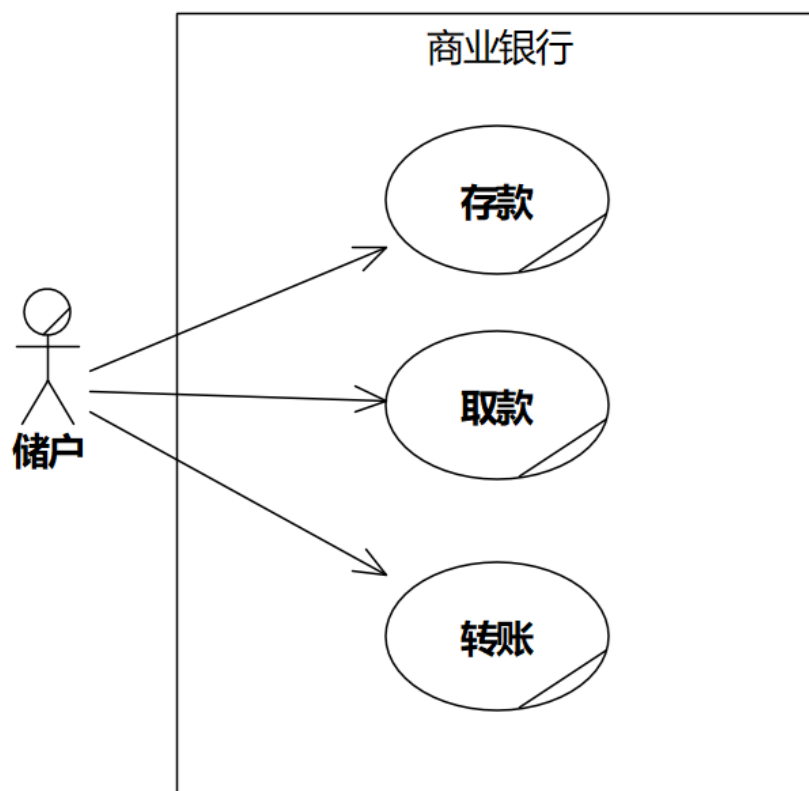


图 3-23 商业银行的用例

（“存款”、“取款”这两个用例，使用频率越来越少了）

组织的用例代表组织的价值，这个价值是很难变化的。如果穿越回一百年前，图 3-23 并没有变化，一直在变化的是用例的实现——业务流程。一百年前，银行要实现“储户→转账”的用例，需要许多人脑系统一起协作，现在则变成了少数人脑系统和许多非人信息系统之间的协作，甚至变成全部是非人信息系统之间的协作，如图 3-24。

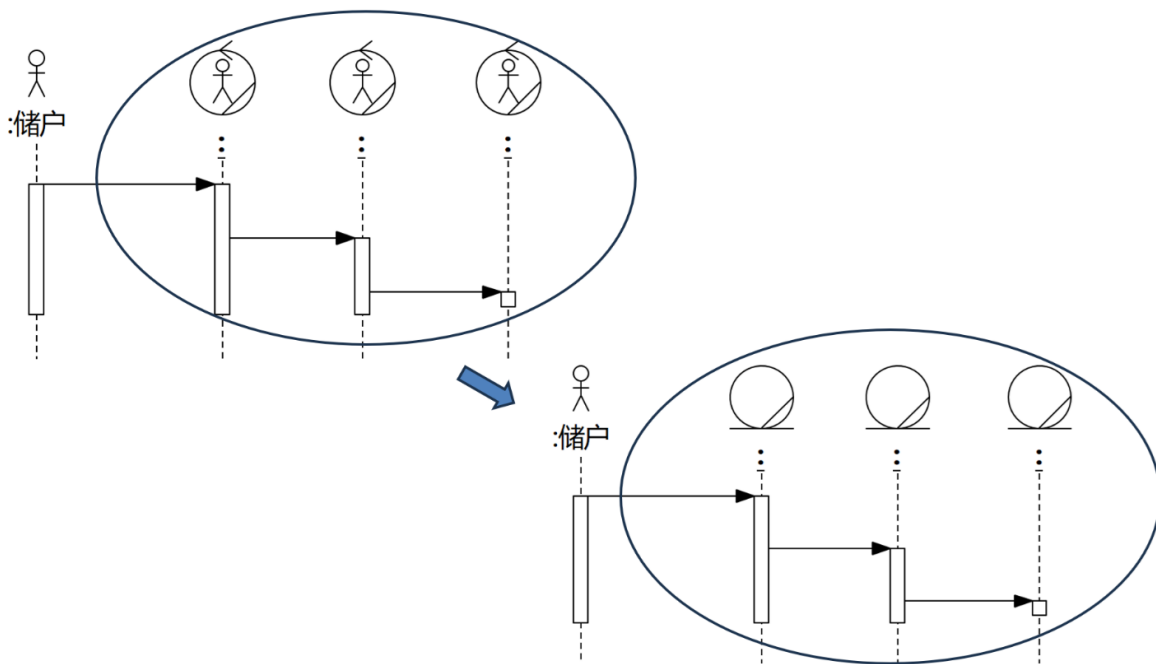


图 3-24 用例没变，实现用例的零件变了

建模时，我们把组织的流程（即所谓业务流程）看作为业务用例的实现，将其归集在业务用例的下面。组织内部之所以有某些流程，是因为要实现组织的某些用例。

把组织里发生的一切都看作为给业务执行者提供价值，这样的思路对改进组织流程有非常大的帮助：先归纳出组织对外提供什么价值，再思考如何更好地优化组织流程和更换组织零件来实现这些价值，如图 3-25 所示。

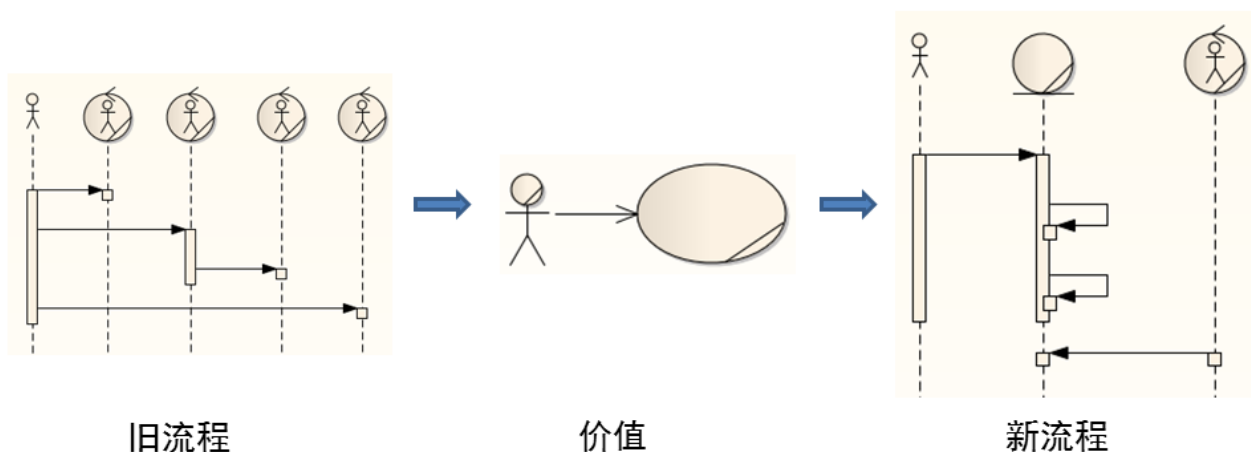


图3-25 从价值出发重新构造组织流程

如果目标系统刚好是所更换的零件之一,重新构造组织流程就可以成为推导需要引进的信息系统的责任——也就是系统需求——的利器。

开发团队经常发现需求“容易变化”。根源之一是需求的来路不正,没有把系统当作一个零件放在组织中来看,而是靠拍脑袋得出需求,导致得到的系统需求是假的,而且不同人、不同时间拍脑袋得到的结果还不一样。

系统投入使用后,发现和组织的其他零件格格不入,又要改。这时,开发团队可能就会发出“需求变化剧烈”的感叹。

如果能正确运用业务建模技能,这些假的“需求变化”就会消于无形。

★不同的庸医给同一患者看病,或者同一个庸医多次给同一患者看病,说辞都不一样,开的药方也不一样——“患者的病情变化太激烈了”!

3.2.4.4 识别业务用例的一些要点

正确理解价值

用好用例,关键在于理解“价值”。价值是**期望和承诺的平衡点、买卖的平衡点**。

例如,以医院为研究对象,“患者→挂号”不是用例,因为挂号不是患者对医院的期望和医院对患者的承诺的平衡点。如果挂到了号后医院的服务到此为止,患者到医院来时心中的期望没有得到满足。或者说,

医院这个研究对象能承诺向患者提供的价值不是挂号，而是看病。

可以这样思考：医院能这样叫卖自己吗——“来呀来呀，我这家医院能挂号呀！”患者一听，“哇，真棒耶，这医院能挂号耶，我赶紧去！”其实患者巴不得不挂号也能看病，只不过人太多了，需要拿号排队。

如果把研究对象改为医院挂号室，“患者→挂号”就是合适的用例。患者对挂号室的期望是能挂到号，不会因为挂号室没帮他看病就破口大骂。挂号室对他的承诺也就是能给他号。

以上提到的正确和错误的用例图，如图 3-26 所示。

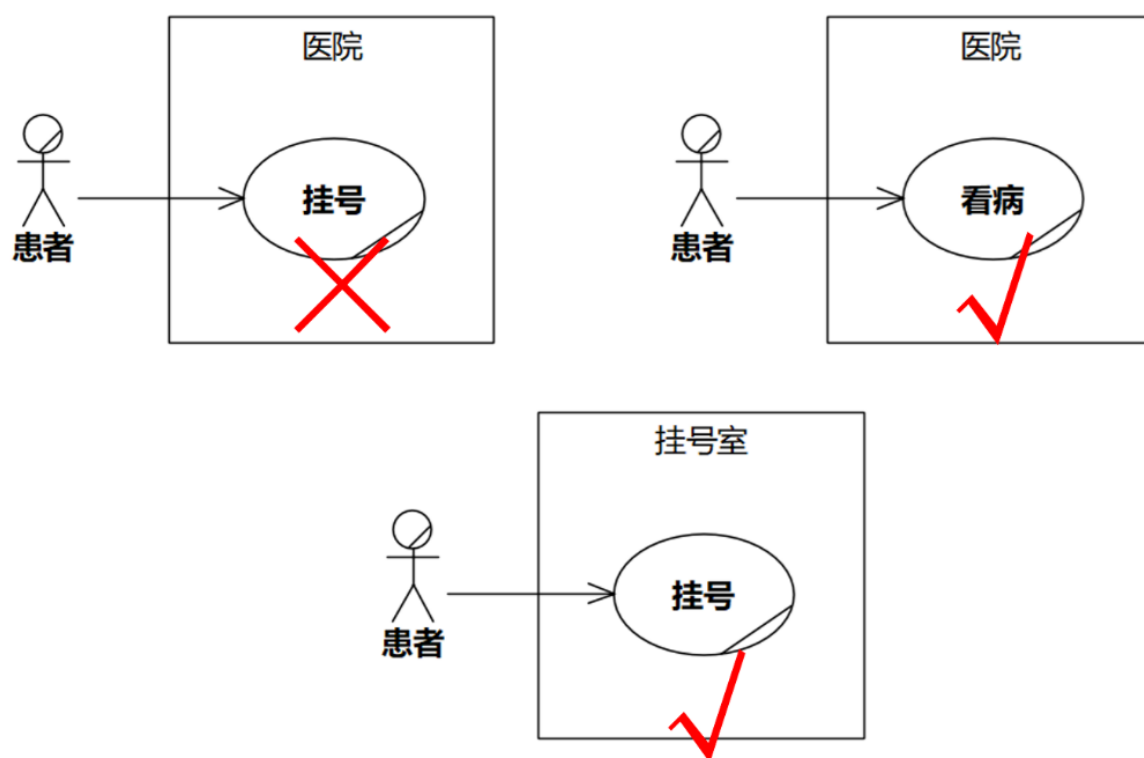


图 3-26 期望和承诺的平衡

如果患者在窗口挂到的是明天的号，他离开医院回家了，明天再来医院就诊，那么挂号算不算医院的一个用例呢？仍然是不算的，因为患者心中仍然在期待，这件事情没有完。

(如果患者在家里使用医院的 app 或小程序挂到了第二天的号,明天再去医院就诊,那么医院的用例图会有变化吗?请读者使用本章前面已经提到的知识点自行解答。)

从以上的思考可以看到,用例引入了人的主观看法。一个组织或系统对外提供了很多交互的接口,为什么这么一段交互刚好是一个用例?这是由涉众的观点决定的。

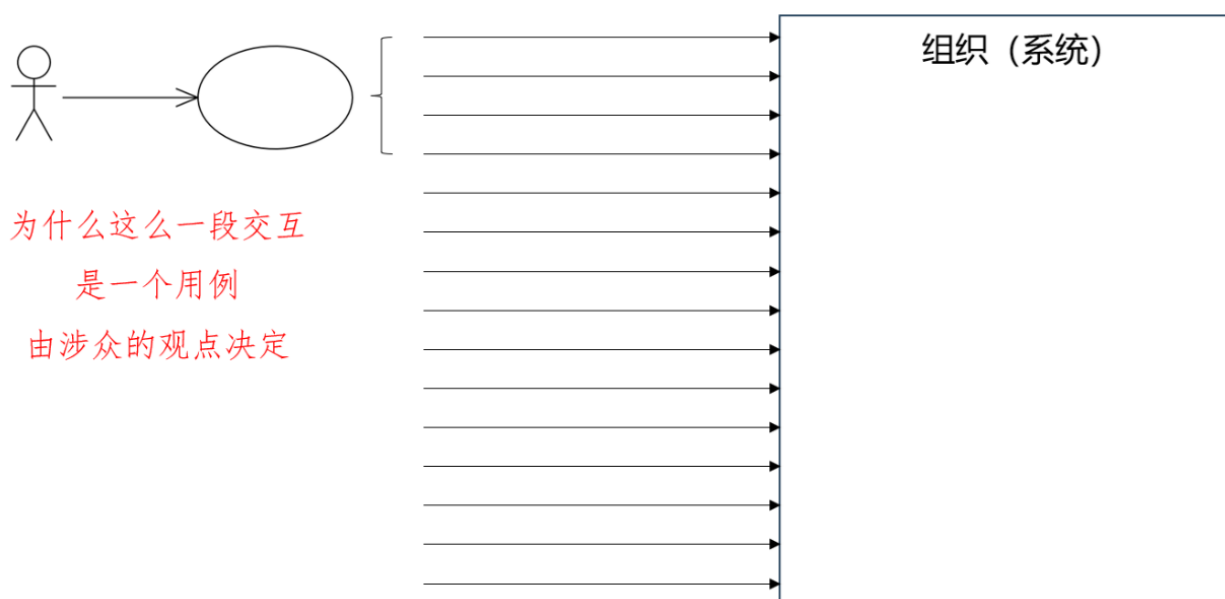


图 3-27 期望和承诺的平衡

你怎么知道涉众是这样想的?那不得去调研涉众嘛!如果你在使用用例的时候没有起到促进你去调研涉众的作用,那就不要装了,干脆就不要用。

可能有的同学还会想:这个世界啥人都有,不同涉众看法是不一样的。例如,我调研患者对医院的期望,没准就有患者说“我的期望就是挂号”,那怎么办?复习第 2 章。

业务用例的时间跨度

业务用例是组织对组织的服务，相对于系统为系统提供的服务（系统用例）来说，所需要的时间是比较长的。

企业和市场监督管理局打交道变更营业执照的过程中，可能要发生多次交互。例如，企业里某个人先在线申报变更，过一些天，预审通过后，企业相关人员各自在线实名认证和签章，算是正式申请，然后再过一些天，领取新执照。

虽然跨度有很多天，而且中间有两段等待的“空白期”，但用例只有一个“变更营业执照”，而不是3个，如图3-28所示。

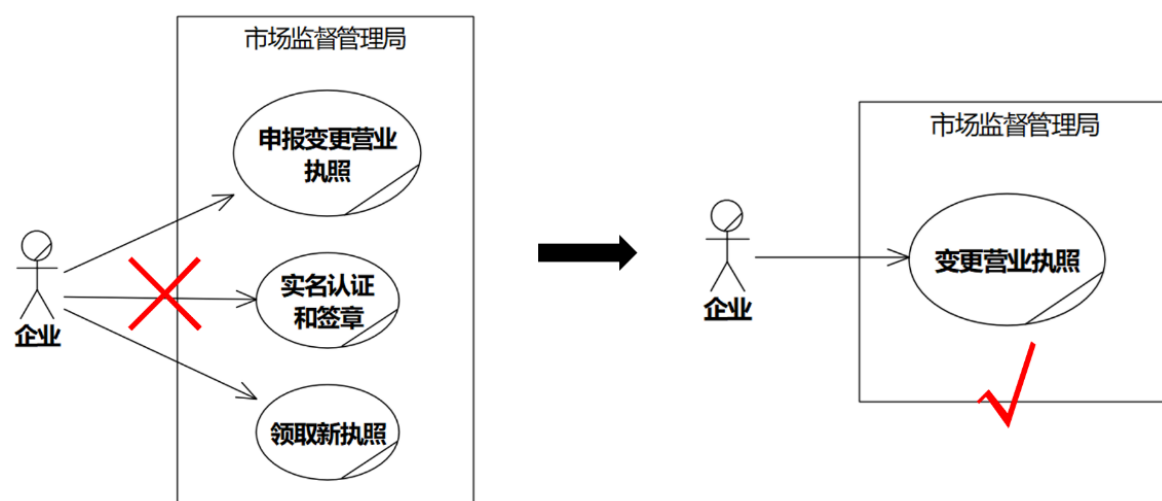


图 3-28 业务用例持续的时间比较长

当然，所谓“空白期”只是企业这边的看法，实际上市场监督管理局正在走流程。不过，即使这段“空白期”什么也没发生，也并不影响什么是用例的判断。例如，刚才说到的患者在网上预约挂号，但预约到的是两周之后的，过了两周才去医院看门诊。

系统用例的持续时间比较短。执行者使用系统做了某事，达到了某个结果，然后离开系统去做别的事情，如果离开时他心里认为得到目前

的结果已经不算白做，就可以把做某事作为一个系统用例。

例如，以“市场监督管理局”为研究对象，像图 3-28 左侧那样给出 3 个用例是错误的。如果把研究对象改成“市场监督管理局系统”（或者名字高大上一点，叫“市场监督管理局一体化平台”之类），图 3-29 就是合适的。

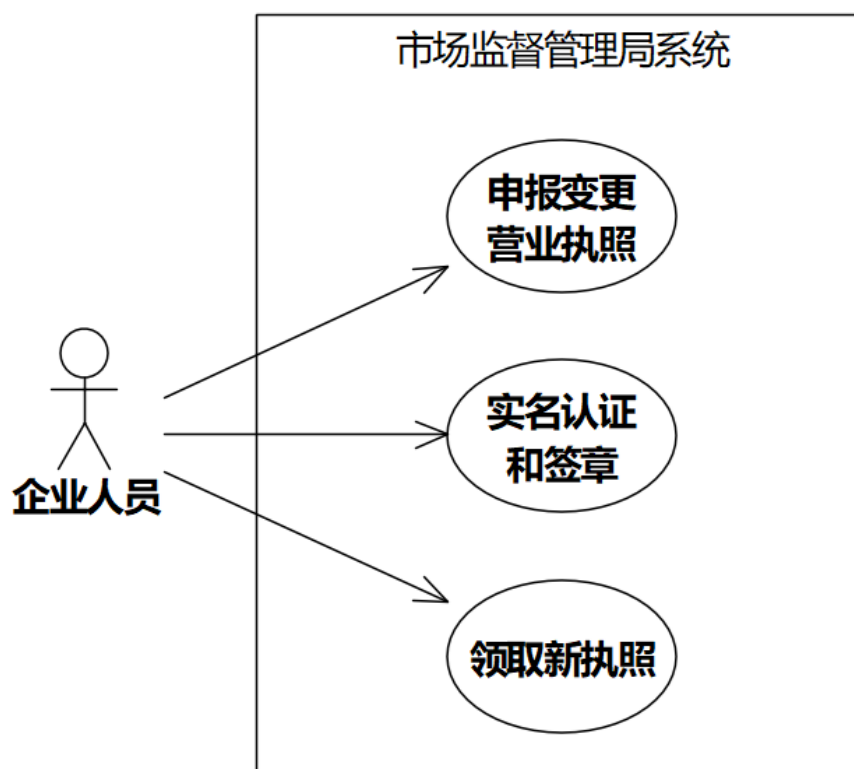


图 3-29 市场监督管理局系统的用例

(1) 此处领取的执照为电子执照 (2) 执行者变成“企业人员”

可能读者会注意到图 3-29 和图 3-28 左侧的高度相似。当前，组织和外部组织交互时，派出与对方交互的系统大部分已经变成了非人信息系统，例如图 3-29 中的“市场监督管理局系统”，因此，这几次交互的目的，就会成为该非人信息系统的用例。

但是要注意，“市场监督管理局系统”很可能还有很多提供给内部

人员使用的用例，例如审批等。说“市场监督管理局系统”提供给“企业人员”的用例和图 3-28 左侧高度相似，是可以的，但不能说“市场监督管理局系统”的用例图和图 3-28 左侧高度相似。

当然,如果将来市场监督管理局的信息化程度发展到变更营业执照不再需要内部员工参与，完全由“市场监督管理局系统”秒审批、秒发证，那么，“市场监督管理局系统”的用例图就和图 3-28 右侧高度相似了，如图 3-30 所示。

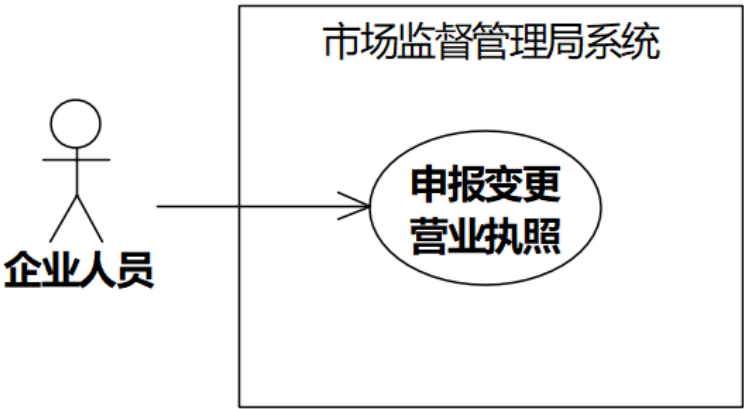


图 3-30 无需人工情况下，“市场监督管理局系统”的用例

有一些词汇带有浓浓的“系统”味道，例如新增、查看、录入、查询、修改、配置.....带有这些词汇的用例，很可能不是组织提供的价值，而是某系统提供的价值。

也可能有这样的组织，例如“**情报所”，它对外提供的价值就是提供一些信息。即使如此，业务用例名字最好也不要“查询**”，这样软件味道十足的名字，可以写成“了解**”。

识别业务用例本来应该是很简单的事情，但是，许多程序员出身的

需求人员受到了以往工作经历的影响，往往把简单的事情变得复杂。下面列举一些常见的错误。

错误一：把内部人员的行为当作业务用例

例如，以医院为研究对象，有人会画出图 3-31：



图 3-31 把业务工人的行为当作业务用例

加上边界框注明研究对象可以减少一部分这样的错误——本来就应该这样做。讨论“是不是用例”、“有哪些用例”的时候，不说清楚研究对象，讨论没有意义。，加上边界框后，建模人员会警觉，收费人员和医生在医院里面，不在外面。如图 3-32。

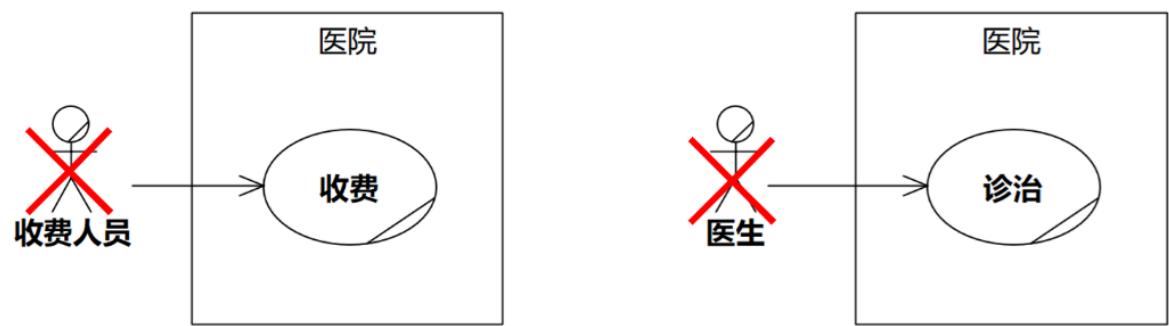


图 3-32 边界框有助于辨别组织内外

有的建模工具没有提供这个边界框，可以用一个 Note 注明研究对象，如图 3-33 所示。

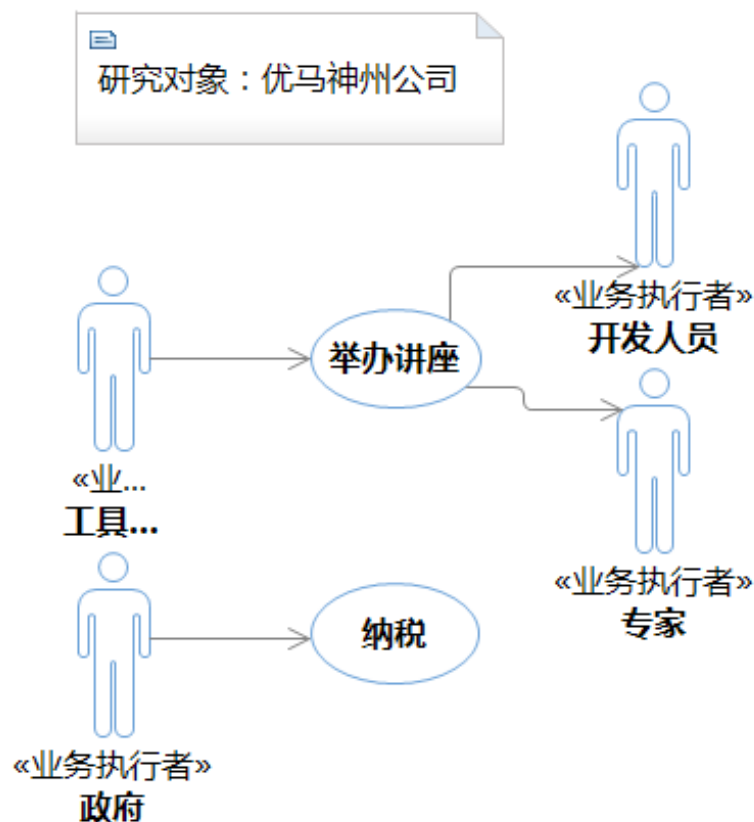


图 3-33 没有边界框，用例图也要用 Note 注明研究对象

也有人觉得没有边界框比起有边界框更能利用更多空间,对比效果如图 3-34 所示。

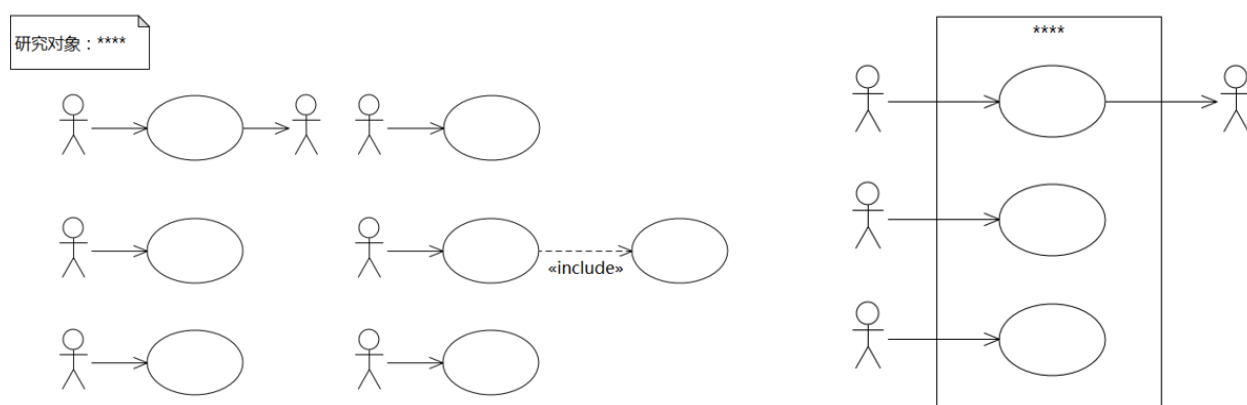


图 3-34 有无边界框的用例图布局对比

我建议初学者还是画边界框,以便时刻提醒自己当前的研究对象是

什么，熟练的建模人员自便。

即使有了边界框，建模人员可能又会害怕，“收费人员→收费”、“医生→诊治”等信息此时不表达出来就忘记了，所以千方百计要赶紧把自己能想到的内容表达出来。他可能会“聪明”地给“收费人员”和“医生”的小人外面加个圈，也就是变成后面要说的业务工人，放在边界框里面，得到图 3-35。

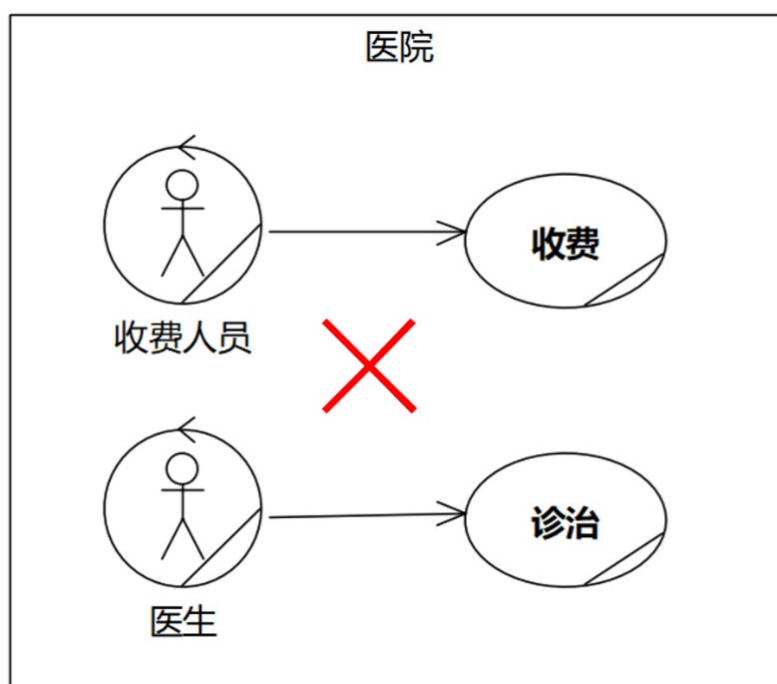


图 3-35 业务工人的“业务用例”，是错的

而且他还有理由：难道医院不要收费和诊治吗？收费和诊治不是医院的本质吗？

这里反映了建模人员常见的一个问题：**分不清问题和问题的解决方案。**

医院确实要收费，但图 3-33 说的不是收费，而是收费的一个解决方案——“收费人员”这个人脑系统来做收费的事情，背后的真实问

题是医院的老板要通过医院来赚钱，至于钱是怎么收上来的，是收费人员这个人脑系统负责的，还是全程无人，完全由非人信息系统互相协作就达到收费的目的，老板是无所谓的。

同理，“医生诊治”也只是解决方案。患者对医院的期望是把病治好，治疗的过程短，不痛苦，没有后遗症，收费还便宜，并不在意他的病是医生动手术治好的，还是外星仪器或灵石给照好的。

医院老板也巴不得不用雇那么多收费人员和医生，照样为患者看病赚钱，只不过目前做不到而已。

或者这样思考，医院的成立不是为了容纳收费人员和医生，以解决本地户口的下岗人员和医科毕业生的就业问题，而是患者要看病、老板想赚钱，于是才有了医院。

像收费人员这样的人脑零件，以现在的 IT 技术替换掉没有问题，不过像医生这样读了很多年书，经过许多年专业训练的人脑零件，替换起来更难一些。

即使现在医生的地位还比较稳固，他的责任也已经被替换了一部分。以前去看病，说“医生我咳嗽”，医生会让我们伸出舌头看一看，听诊器放胸口听一听，躺在床上按一按。现在呢，医生抬头看一眼，啪啪就开单，“去照个××吧”，把检查的责任转移给仪器了。

尤其最近几年 AI 大发展，目前 AI 的能力很可能已经超过了平均水平的医生。

这里反映了建模人员常见的另一个问题：**害怕忘了，想到什么赶紧**

就写上。

没有掌握建模方法学的建模人员,既没有能力也没有自信按照方法学的思考过程,像坦克一样一步步推过去。一旦他“灵机一动”想到什么,就会觉得很宝贝,害怕再不记下来一会就忘了。

所以,不管当前在做什么,他都会不计一切代价赶紧把自己的“灵感”记下来,因为这已经是他的“巅峰”了,忘了损失就大了。即使他目前在画组织的流程,当他想到某个代码的 tips 时,他也会把它塞进当前的工作中。

错误二：业务用例随待引入系统伸缩

有的建模人员把臆想的待引入系统的用例直接当成组织的用例画出来。如图 3-36 所示,建模人员要做一个医院信息系统,他先臆想,医院信息系统有一个“护士→核对医嘱”的用例,在被问及医院的用例的时候,他只需要把“医院信息系统”改成“医院”就行了。

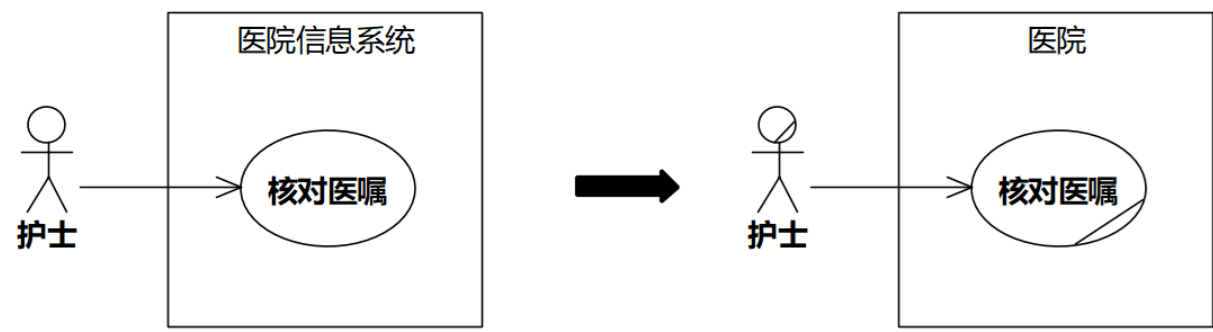


图 3-36 将臆想的系统用例当成组织的用例

根据前面讲的知识要点,一看图 3-36 右侧,护士在医院这个组织的边界外面,就知道不对了。但是,要求建模人员按照业务用例的定义做时,有人就会说:我的系统就是这个功能,我已经知道了,我还要考

虑其他东西干什么？

这是一种“致命的自负”。正是因为很多情况下拍脑袋得到的是错误的需求，所以才要做业务建模，从组织的价值来推导系统应该具备什么价值才会对组织有帮助，这样系统才能卖得出去。如果已经认定了系统有这些功能，直接画系统用例图不就完了吗，还装模作样做业务建模干什么呢？

好了，现在建模人员知道护士在所研究组织边界里面，不能作为业务执行者了，但是又有可能还是受待引入系统的影响，导致组织的价值随着所引入系统的价值大小伸缩，如图 3-37 所示。

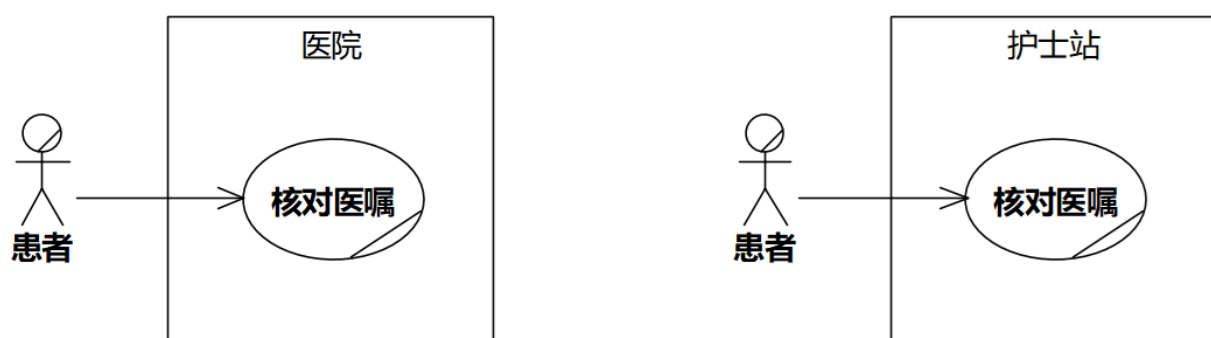


图 3-37 组织的价值受所引入系统影响

这一次，执行者不再是护士了，而是患者，这回应该合格了吧？还是不对！建模人员还是惦记着他所臆想的系统功能“核对医嘱”，就像前面所说的，把它当成宝贝一样，害怕这个时候不写上，自己“巅峰状态”的“灵感”就浪费了。

所以图 3-37 中，医院或护士站就变成了一个为患者提供核对医嘱服务的机构。

碰到这种情况，我通常都会用开玩笑的口气说，幸亏你们团队不是

做马桶的，否则就有可能得到图 3-38。

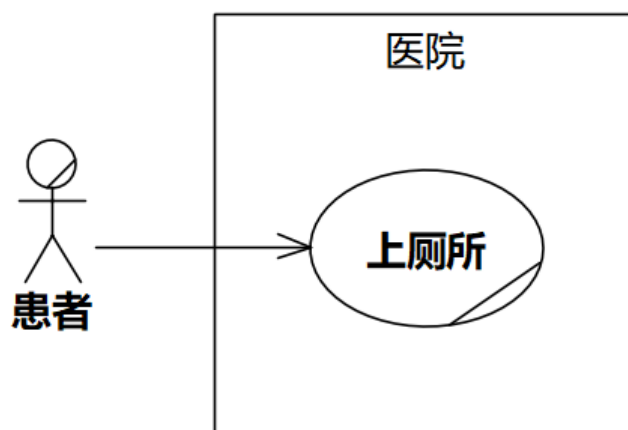


图 3-38 医院变成了上厕所的机构

即使真的是卖马桶的，想要打败其他对手把马桶成功卖给医院，也依然需要研究医院的流程，找到适合用马桶来改进的改进点，才能打造出为医院量身定制的贴心马桶，如图 3-39 所示。

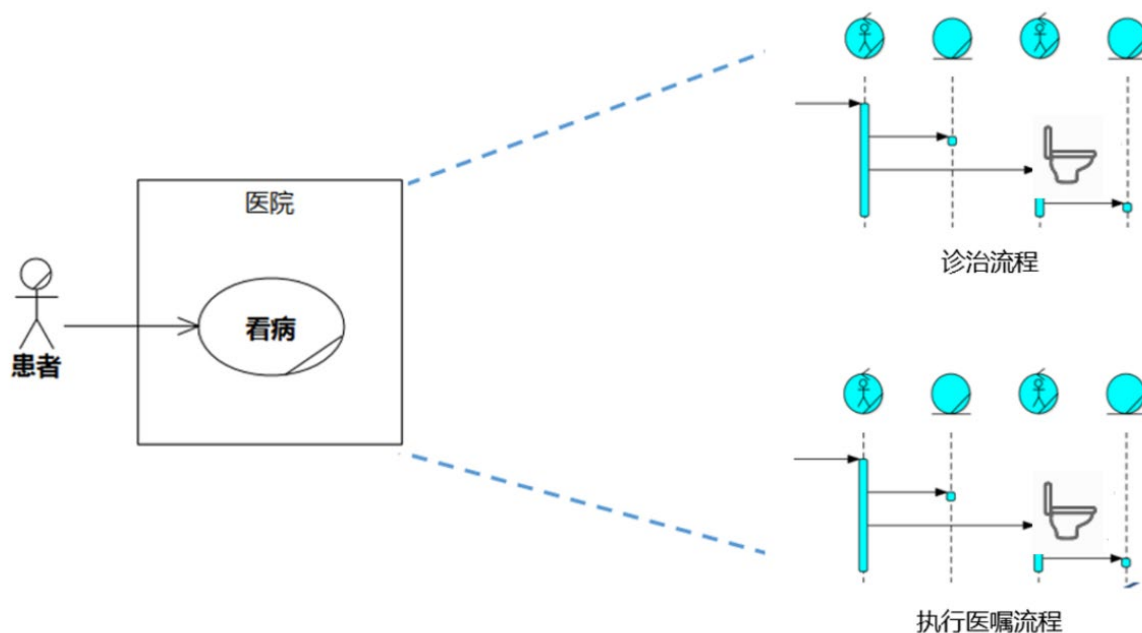


图 3-39 马桶也要从医院的流程找需求

一个组织，甚至组织的一条流程都涉及许许多多的系统。有可能在开发不同的系统时，研究业务用例和业务流程，发现得到的结果和开发另一个系统时的结果一样。

这是很正常的，因为所研究的就是同一个组织甚至同一个流程。建模人员不必因此感到惊慌，更不要因为“业务用例一样”、“业务用例太少”、“业务用例太简单了”，不自觉地要“创新”，把自己臆想的待引入系统的用例搬上来。

错误三：为支撑流程捏造业务用例

以医院为研究对象，识别出用例是“患者→看病”。如果建模人员判断要做的系统和患者挂号、门诊、收费等流程相关，他会比较安心，因为预计往下建模时，他想要看到的部分会出现。

如果要做的系统和药剂科的药师盘点药品的流程有关，建模人员可能就会慌张，因为这个看起来好像不能从患者看病的流程里推导出来，于是又开始害怕自己的“灵感”没了，非得想办法给它捏造一个业务用例的身份，如图 3-40。

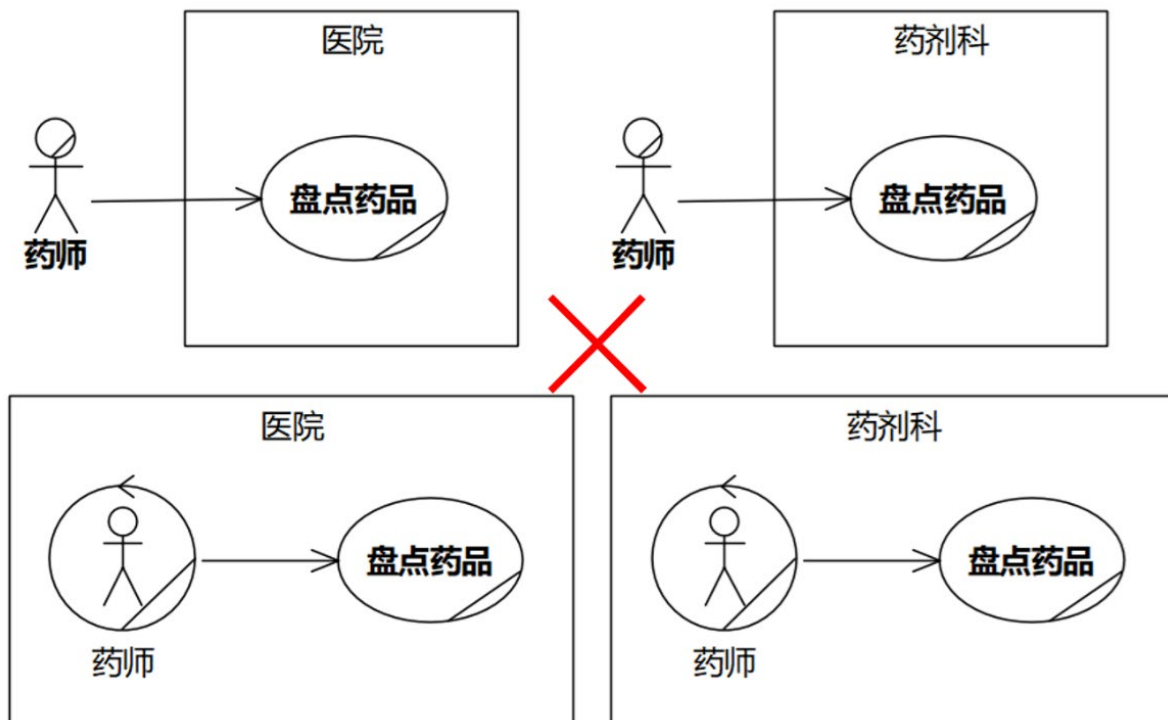


图 3-40 非得把“盘点药品”弄成业务用例

图 3-40 中的所有做法都是错误的。上面两个，不管以医院为研究对象还是以药剂科为研究对象，药师都在外面。前文说到图 3-35 时已说过，下面两个也是不行的。

“盘点药品”不是用例，以医院为研究对象时，它可以作为“患者→看病”用例的支撑流程；以药剂科为研究对象时，它可以作为“患者→取药”用例的支撑流程。如图 3-41 所示。

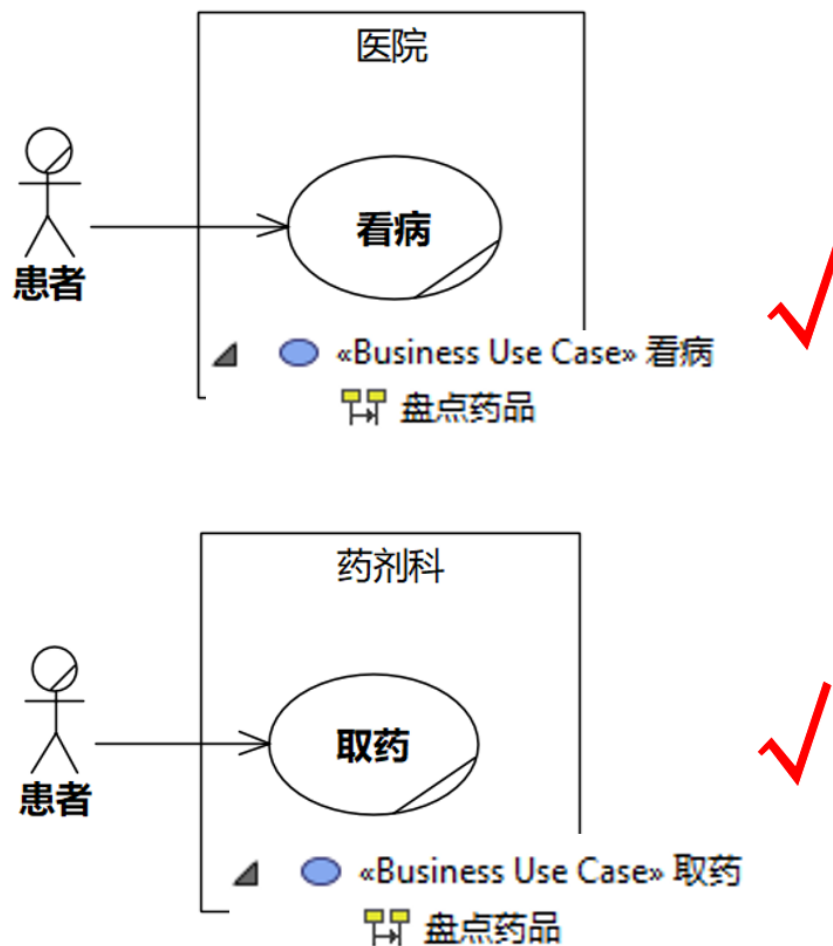


图 3-41 用例下面的支撑流程

如果药师不盘点, 会导致真实库存和账面库存有差距, “患者→看病”的主流程进行到取药片段时, 才发现其实某个药品已经没有库存或者现有库存药品是坏的, 往“患者→看病”的成功目标行进的道路上遇到了阻碍。

同理, 医院里为什么有保洁员打扫卫生? 为什么内部要花时间搞团队建设? 都应该从“患者→看病”的高度来看。

把研究对象缩小到“药剂科”时, 这个思考也是相似的。

另外一种捏造的做法是, 为它捏造一个“管理型业务用例”, 如图 3-42:

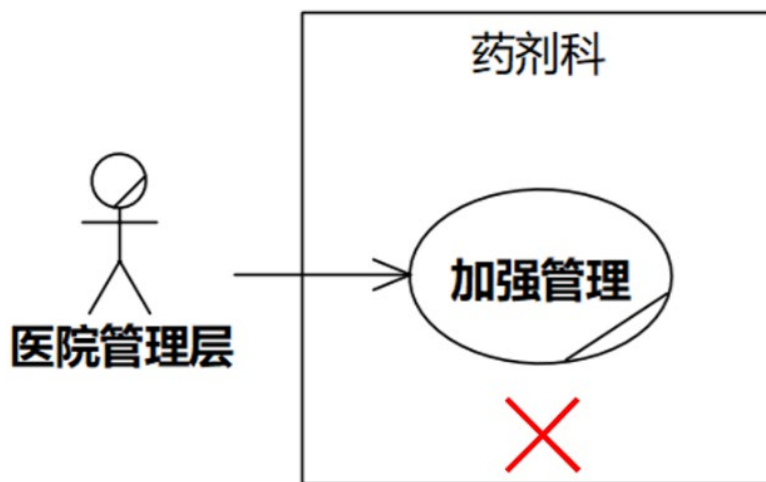


图 3-42 捏造“管理型业务用例”

建模人员的思路可能是这样的：药师为什么要盘点药品？因为医院管理层要加强管理！看起来好像可以，但这是放之四海皆准的废话。接下来再问，为什么要加强管理？还是得追溯到为外部提供的价值。

3.2.5 自测题



本书不提供习题答案，请扫码或访问 umlchina.com/book/quiz3_1.html 完成在线测试，做到全对，自然就知道答案了。



1 [单选]

“业务”是一个模糊用语。在各种带有“业务”的用语中，其中“业务”的含义会有不同。

以下用语中，其中“业务”的含义和其他几个有别的是：

☐A) 业务规则

☐B) 业务流程

☐C) 业务工人

☐D) 业务用例

2 [单选]

曾经有新闻报道“大熊猫玩菜刀”，大熊猫“萌萌”在活动场内捡到了一把菜刀，双手抱着把玩，甚至一度将刀刃贴在脸颊上。

如果动物园决定引入 IT 系统，监控并辨别大熊猫的危险行为，一旦判断出有类似“玩菜刀”之类的行为，就向动物园熊猫馆管理员报警。

以下说法正确的是：

☐A) 在业务建模工作流，研究对象应该定为熊猫馆

☐B) 在业务建模工作流，研究对象应该定为大熊猫

☐C) 在业务建模工作流，如果大熊猫是一只，研究对象应该定为大熊猫，如果大熊猫是多只，研究对象应该定为熊猫馆管理员

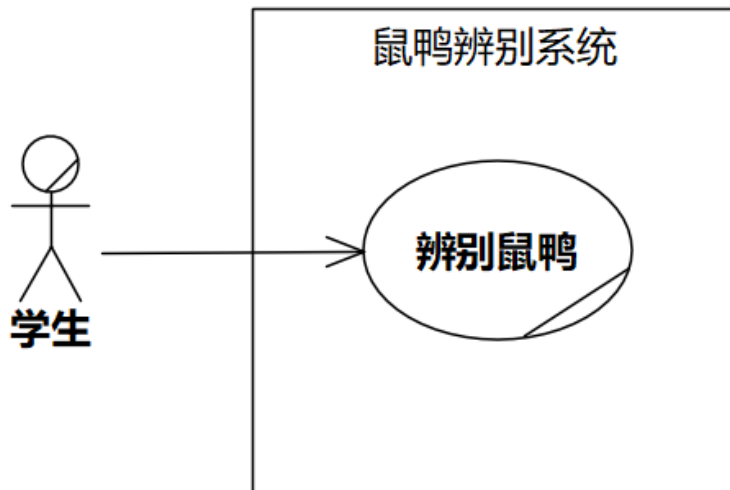
☐D) 在业务建模工作流，研究对象应该定为熊猫馆管理员

3 [单选]

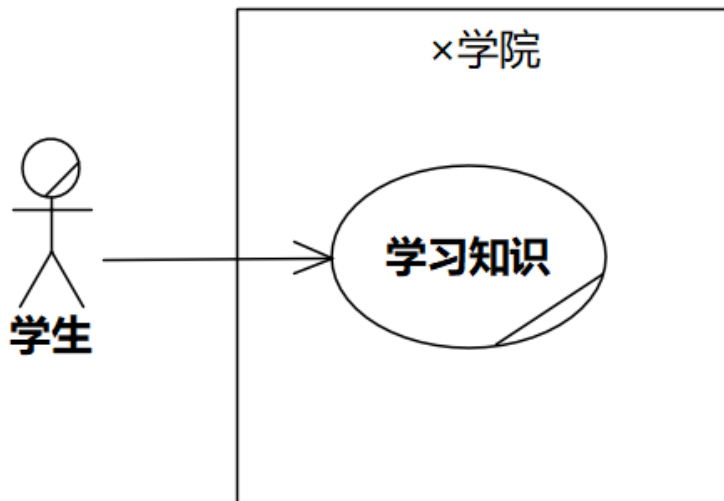
以一款“鼠鸭辨别系统”为目标系统,现在思考业务建模工作流的业务用例图。

以下是不同的建模人员所画的“业务用例图”,可以判断必定错误的是:

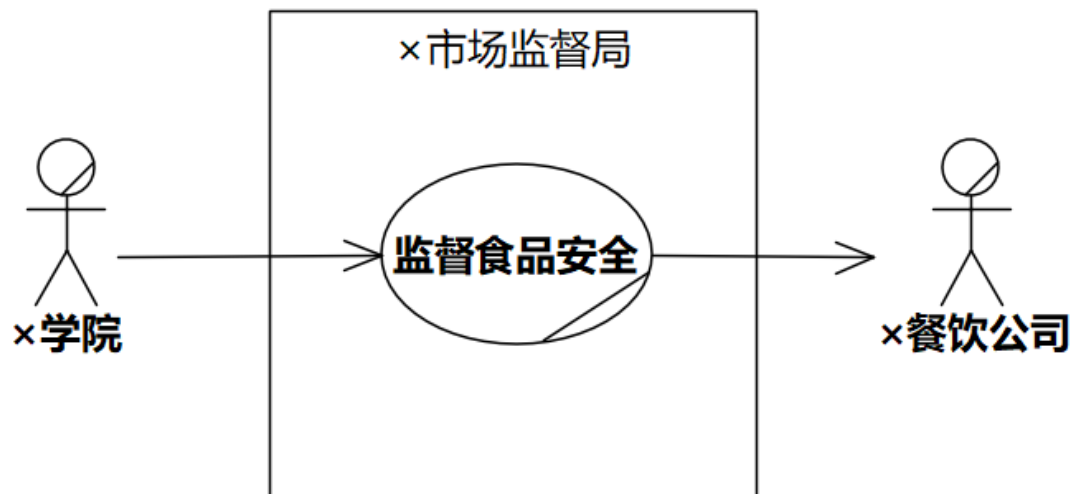
☐A)



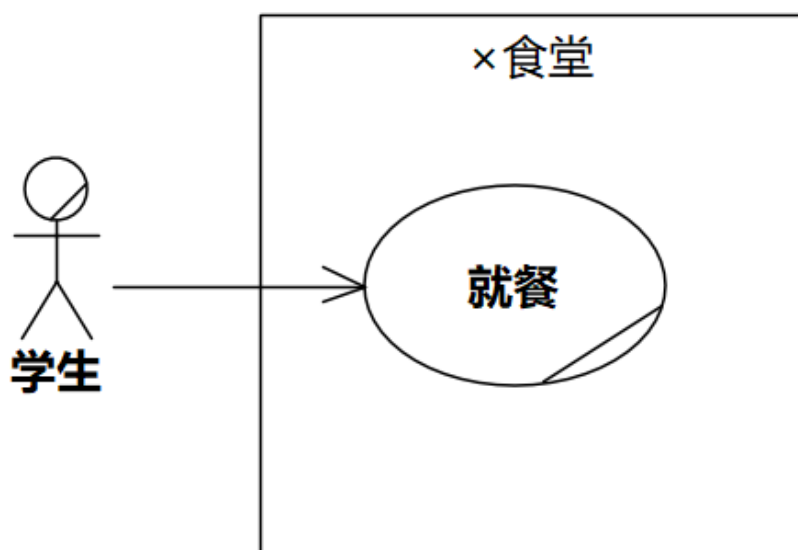
☐B)



☐C)

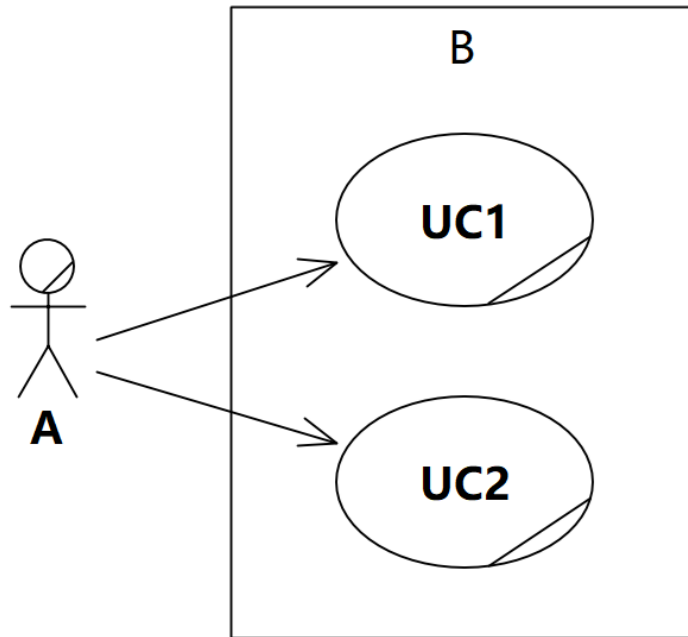


□D)



4 [单选]

张三本来想画这样一个用例图：



后来发现所用的工具中，工具箱里没有带斜杠的小人和椭圆，也无法通过设置小人和椭圆的属性达到。

张三决定用不带斜杠的小人和椭圆表达。此时，张三还需要在模型中添加什么信息，就能达到所给图形表达的效果。

☐A) 给不带斜杠的小人添加<<Business Actor>>文本构造型

☐B) 给不带斜杠的椭圆添加<<Business Use Case>>文本构造型

☐C) B 是一个组织

☐D) 采用领域驱动设计的限界上下文和通用语言业务用户领域需求架构建模法，令某个上下文中的所有小人含义为 Business Actor，所有椭圆含义为 Business Use Case

5 [单选]

针对以下研究对象，“财务人员”最有可能是执行者的是_____。

☐A) 某省注册会计师考试委员会

☐B) 某市税务局

☐C) 某公司人力资源部

☐D) 某公司财务部

6 [单选]

关于业务用例和系统用例的区别，以下说法正确的是：

☐A) 业务用例研究人工，系统用例研究自动化

☐B) 业务用例研究组织，系统用例研究系统

☐C) 业务用例研究业务，系统用例研究技术实现

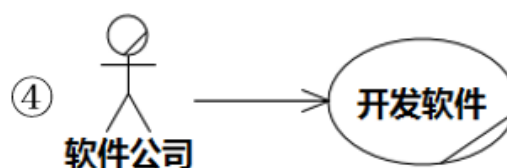
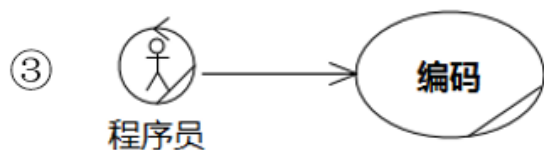
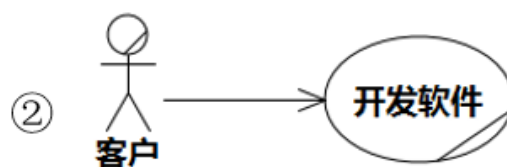
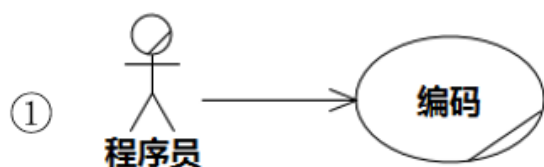
☐D) 业务用例研究系统外的的工作，系统用例研究系统负责的工作

☐E) 业务用例抽象，系统用例具体

☐F) 业务用例不是所有系统都有，系统用例所有系统都有

7 [单选]

以一家软件公司为研究对象，以下正确的是



☐A) ②和③

☐B) 只有④

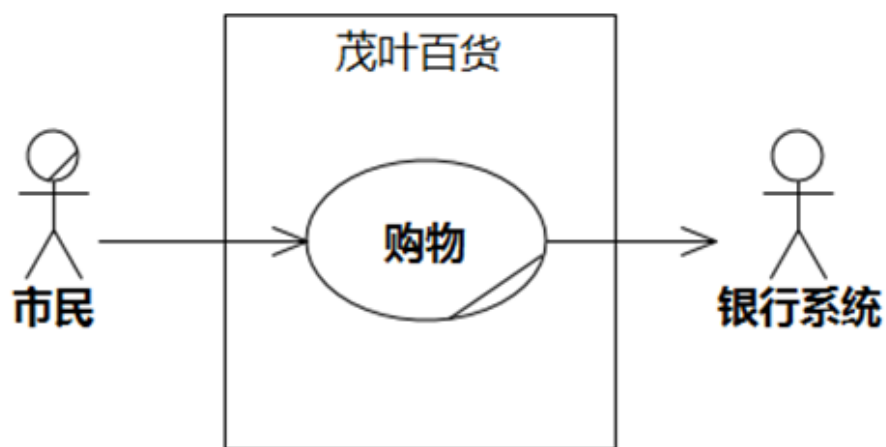
☐C) 只有②

☐D) ①和④

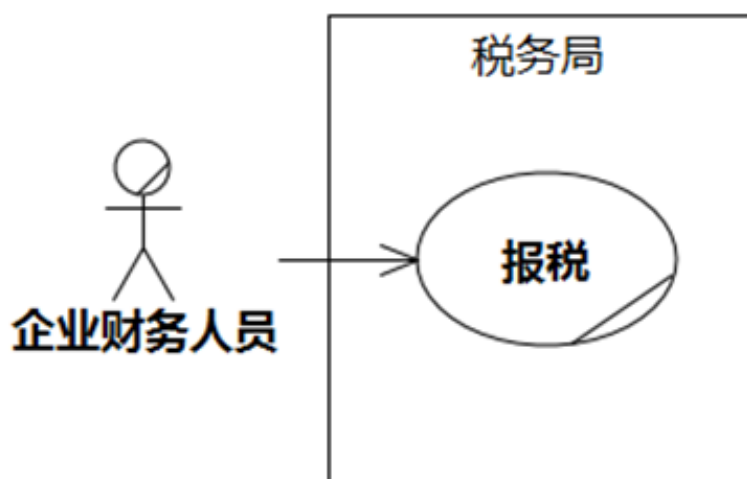
8 [单选]

以下业务用例图正确的是：

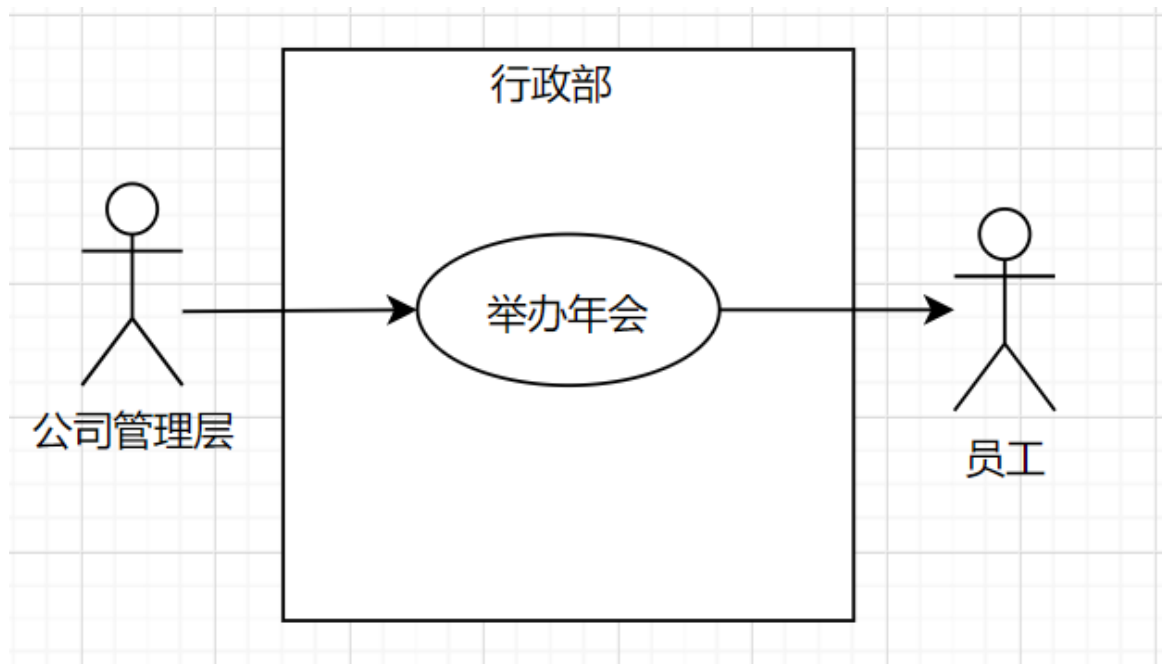
☐A)



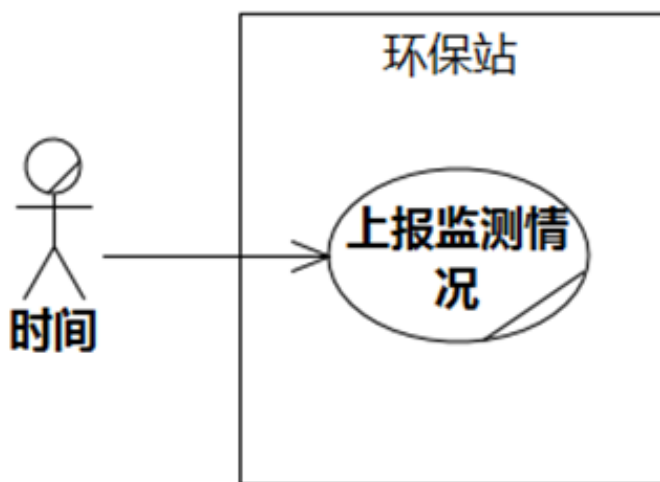
☐B)



☐C)



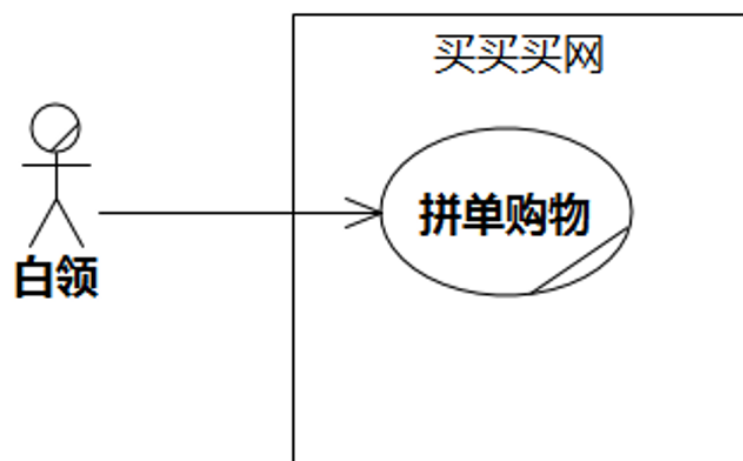
☐ D)



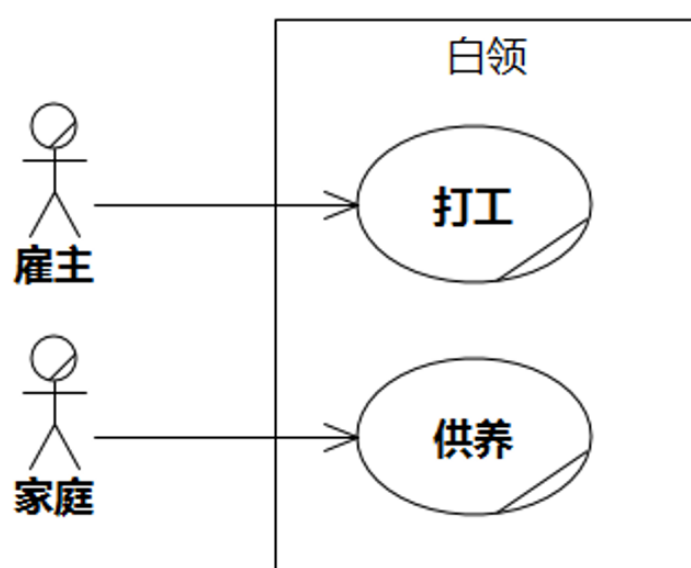
9 [单选]

某人成立了“买买买公司”，要运营一个针对白领的网站（假设名字叫“买买买网”）来挑战京东、拼多多，想通过业务建模来获取网站的需求，以下哪一个业务用例图最合适？

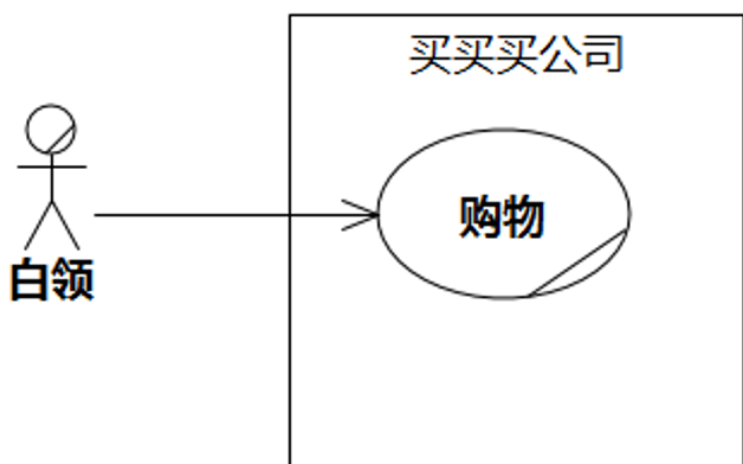
☐ A)



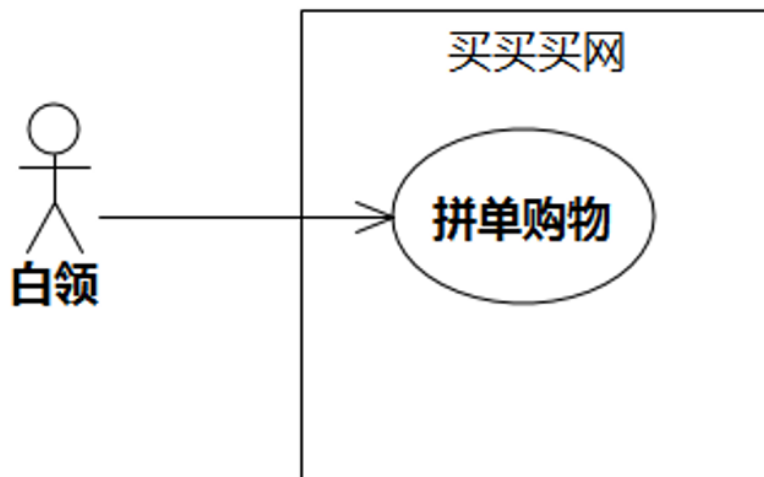
□B)



□C)



☐D)



10 [单选]

以某机关单位的 OA（办公自动化）系统为研究对象，关于该系统的业务执行者和业务用例，以下说法最合适的是：

☐A) 其中一个业务执行者是员工，其中一个业务用例是发公文

☐B) 其中一个业务执行者是员工的代表人物——办公室主任马宝国，其中一个业务用例是发公文

☐C) 题目没说清楚，有更多细节才好确定该系统的业务执行者和业务用例

☐D) 不存在

11 [单选]

公交公司里有调度员，调度员的工作除了调度之外，还要制定线路行车作业计划，还要不定期上路调查客流等。假设根据愿景判断，下一步改

进点应该在调度员上路调查客流的环节,那么这个环节应该归属哪个业务用例呢?

①以公交公司为研究对象的“市民→乘车”用例

②以公交公司为研究对象的“调度员→调查客流”用例

③以系统为研究对象的“调度员→调查客流”用例

④以调度室为研究对象的“公司管理层→调度”用例

⑤以公交公司为研究对象的“公司董事会→提高运营效率”用例

☐A) ①和④

☐B) 只有③

☐C) ②和⑤

☐D) ③和⑤

3.2.6 实作步骤 A-3-1 和 A-3-2：识别业务执行者和业务用例

延续[2.5.3 实作步骤 A-1-3 定位目标组织和目标组织负责人]的结果,目标组织为“马宝中个人组织”,针对它画用例图如图 3-43:

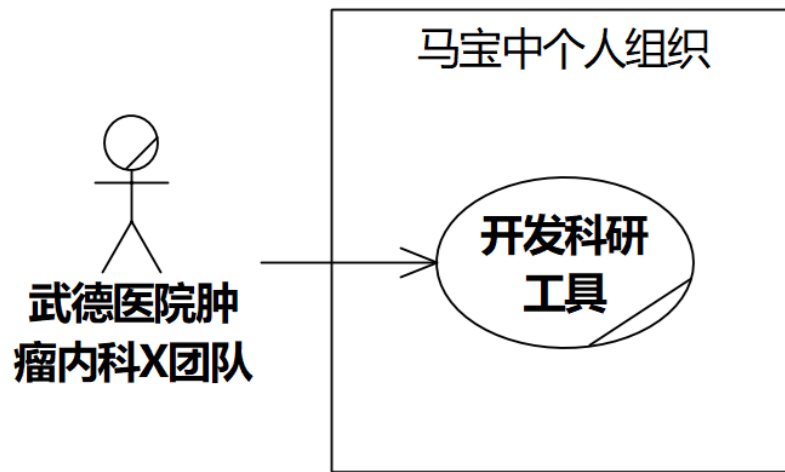


图 3-43 “马宝中个人组织”的用例图

得到这张图，经历了两个定位思考：

- (1) 从“发糕”定位其目标组织为“马宝中个人组织”；
- (2) 从“马宝中个人组织”的用例定位具体的执行者“武德医院肿瘤内科 X 团队”。

用类图建模目前所涉及到的用例相关概念，如图 3-44：

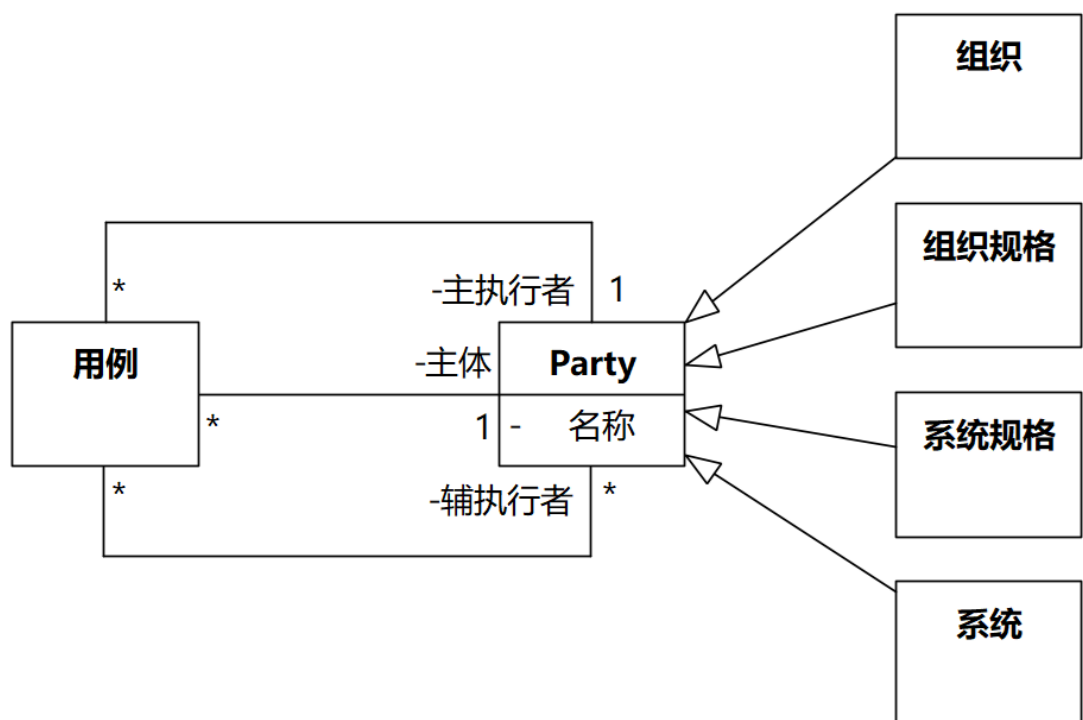


图 3-44 用例相关概念

图 3-44 中, 引入了 Party 的概念。根据前文, 用例可以表达为“组织规格为组织规格提供”、“组织为组织规格提供”、“组织为组织提供”、“系统规格为系统规格提供”等。引入 Party 后, 可以表达为“Party 为 Party 提供”。

★在 Hay 的 Data Model Patterns 和 Fowler 的 Analysis Patterns 中, 使用 Party 作为人和组织的泛化, 此处把它扩展到非人信息系统。

★以下这一段内容阅读起来有一定难度, 可以跳过

如果有读者阅读过 UML 规范, 可能会有困惑。

UML 规范中, 用例部分的 Abstract Syntax(抽象语法)如图 3-45:

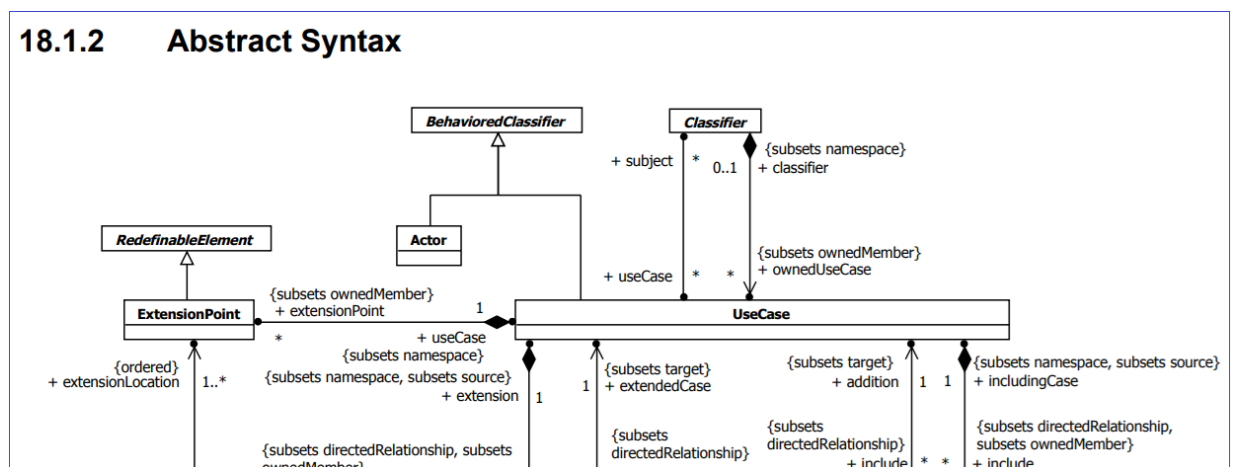


图 3-45 OMG UML 2.5.1 规范 Use Case 部分的抽象语法

图 3-45 和图 3-44 的相同之处是 subject (主体) 作为角色出现;

图 3-45 和图 3-44 的区别是:

UseCase 的 subject 角色所对应的类, 图 3-44 中是 Party, 图 3-45 中是 Classifier;

图 3-44 中, Actor (执行者) 表示为 Party 在用例中扮演的角色, 图 3-45 中, Actor 和 UseCase 都是 BehavioredClassifier 的子类, 而 BehavioredClassifier 又是 Classifier 的子类 (图上未显示), Actor 和 UseCase 之间的关联通过 Classifier 之间的 Association 体现 (图上未显示)。

图 3-44 可以看作本书方法学的领域模型, 不等同于 UML 表示法的元模型, 只是本书方法学在很多地方使用了 UML 中的概念以及表示法, 所以两个模型会有重叠之处。

接下来的内容还会有类似重叠的地方, 这是因为本书决定把封装本书方法学的“发糕”智能需求和设计建模工具作为案例。如果把案例换成第 2 章提到的“智慧烟叶种植管理系统”或“智慧公寓系统”, 类似的重叠和读者的困惑可能就消失了。

图 3-43 中, “马宝中个人组织”是“主体 (subject)”, “武德医院肿瘤内科 X 团队”是“执行者 (actor)”, 这是在用例“开发科研工具”中扮演的角色。

同样, 我们也可以针对“武德医院肿瘤内科 X 团队”画用例图, 如图 3-46, 此时, “武德医院肿瘤内科 X 团队”就成了“主体 (subject)”。

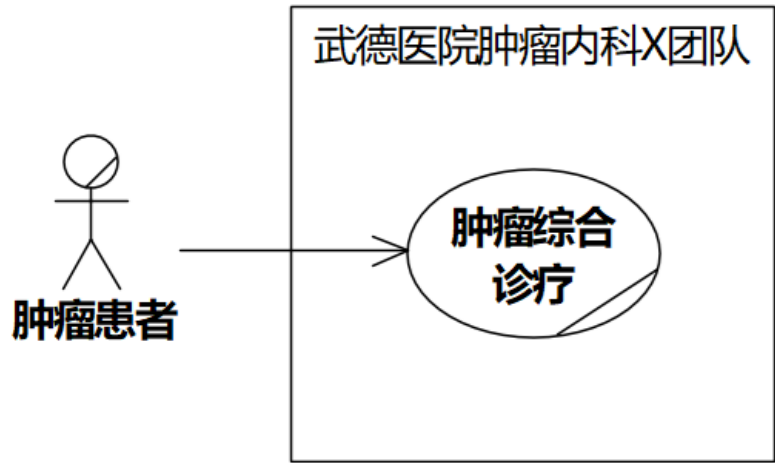


图 3-46 针对“武德医院肿瘤内科 X 团队”画用例图

用对象图表示“发糕”案例目前涉及的系统规格和组织如图 3-47：

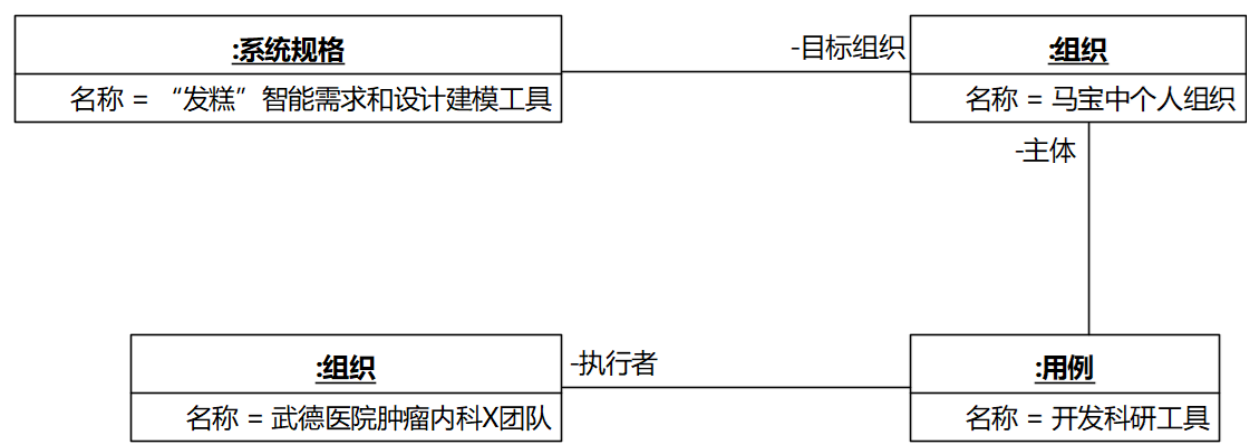


图 3-47 用对象图表示“发糕”案例目前涉及的系统规格和组织

第8章 分析之分析类图——知识篇

墙上挂了根长藤，长藤上面挂铜铃

《长藤挂铜铃》；词：元庸，曲：梅翁（姚敏），唱：逸敏，1959

从分析 workflow 开始，我们每个内容都分为两章。一章讲述建模知识，一章讲述建模知识如何应用在本书案例中。这样的分割主要考虑到更符合实际的工作。

例如，在讲解分析类图时，我们讲解知识的顺序是这样的：

- (1) 识别类和属性
- (2) 审查类和属性
- (3) 识别类之间的泛化
- (4) 识别类之间的关联

如果把案例剖析分解到每个知识点，为了让案例的剖析符合内容的顺序，可能就会出现这样的情况：

讲解完识别类和属性后，案例剖析时，先列出很多类和属性，但没有泛化和关联关系，因为类的关系还没有讲到，所以即使观察到，也故意不画上去；接下来，讲解完类的关系后，案例剖析时，再把关系加上。

这不符合实际工作中的情况。实际工作中，以上列出的几项工作看起来是交叉进行的。

我们在识别类和属性的过程中，可能会发现一些类有相同的属性，

于是泛化出超类，可能会发现有的属性还可以在同一领域内分解，于是识别出另一个类以及两个类之间的关联.....

注意我的用词：【看起来】交叉进行，因为更微小的过程依然遵守上述的推导过程 (1) → (2) → (3) (4)。

为了避免造成误解，我们先完整地讲解知识部分，本书案例如果有和所讲解知识相关的内容，会随时引用。然后在案例部分，按照实际工作中的思考方式灵活应用前面所讲解的知识点。

8.1 分析 workflow 概述

8.1.1 知识的表达和组织

在业务建模和需求 workflow，我们一直把目标系统看作是一个整体，想办法推导出涉众在意的整体表现，即系统的需求。

系统为了满足需求，必须封装一定的知识。这些知识不会乖乖地自己从知识的海洋中走出来，而是需要软件开发人员一点点识别出来，并
有组织、有次序地把它们封装到系统中。

因此，我们要思考：

(1) 如何准确表达系统需要封装的知识，让系统满足需求；

以及

(2) 如何合理组织系统这些知识，低成本地让系统满足需求。

如果不能合理组织知识，当新需求到来时，准确表达知识的成本也

会越来越高。如果考虑到利润，很难停留在（1）而不追求（2）。

不管是纯粹在大脑里面打转转，还是借助了纸笔或建模工具来协助，以上的思考都是逃不掉的。

如果需要封装的逻辑很简单，人脑的容量和运算速度能够胜任，在大脑里打转转还可以勉强应付，但是，能带来利润的系统都是复杂的（参见第 1 章），借助纸笔或建模工具来显式表达思考的过程很有必要，毕竟大脑容量和运算速度比一般人高出一个数量级的天才是很稀罕的。

注意区分“在大脑里完成”和“拍脑袋”的区别。

有的人故意不显式表达，声称“大脑思考就够了”，**背后的真相可能不是天才而是遮羞**——你让他显式表达，他也表达不出来，因为他没有掌握准确表达知识和合理组织知识的方法。

就像考试一样，面对一道有难度的填空题，考生可能有以下三种表现：

学霸：在大脑中迅速完成推导，直接在答卷填写结果。

普生：拿出草稿纸，规规矩矩推导，然后在答卷填写结果。

学渣：拍脑袋蒙一个，直接在答卷填写结果“敏捷试错”。

因为学霸的“在大脑中推导”和学渣的“拍脑袋”有一个容易观察到的共同点：没有草稿纸，所以学渣有时候会用这个共同点来假装自己是学霸，这一点要警惕。

8.1.2 核心域和非核心域

一个信息系统封装了若干领域的知识。其中，有一个领域的知识是该系统不能抛弃或替换的，这个领域称为"核心域"，其他领域称为"非核心域"。

图 8-1 展示了不同系统类型的核心域和非核心域概念。

系统类型	核心域概念	非核心域概念（选择之一）
文档处理器	文档、页、行、字.....	CStringArray、CFileDialog、MSXML.....
电子商务网站	商品、订单、会员.....	</div> 、 ActionForm 、 SessionFactory.....
操作系统	处理器、内存、页面.....	struct、char、int

图 8-1 不同系统类型的核心域、非核心域概念

以文档处理器为例，开发 Microsoft Word 和 LibreOffice Writer 所使用的编程语言和组件不一样，但文档、页、行、字等核心域概念是一样的。即使回到计算机诞生之前或者去到未来，这些概念也依然存在。

还需要注意的是，核心域既可以是非计算机领域，也可以是计算机领域。例如，图 8-1 中的操作系统，其核心域概念有“处理器”、“内存”等。

关于“核心域”和“非核心域”，一种常用的通俗说法是“业务”和“技术”，但“业务”和“技术”的说法不严谨。

有的开发人员在潜意识里用“懂”、“感兴趣”来划分“业务”和“技术”：

*我懂且我感兴趣的知识→技术；（我懂 Java 编码，我对 Java 编码感兴趣，Java 编码是技术）

*我懂但不感兴趣的知识→业务；（下单、收银、配送我懂一些，但不感兴趣，这些是业务）

*我不懂但感兴趣的知识→高科技；（我不懂深度学习，但很感兴趣，哇塞，高科技）

*我不懂且不感兴趣的东东→忽悠。（我不懂 UML 建模，也不感兴趣，妈的，忽悠）

有的开发人员在潜意识里则用“是否和计算机有关”来划分“业务”和“技术”：

*和计算机无关→业务；

*和计算机有关→技术；

【说明 1】

本书中的“核心域”和 Eric Evans 以及 DDD（领域驱动设计）话语体系中的“核心域”（Core Domain）意思不同。

本书中的“核心域”指信息系统中不可替代的那部分内容——这个以软件开发人员的知识是可以判断的。

DDD 话语体系中，把“领域”（相当于本书中的“核心域”）划分

为“核心域”、“通用子域”、“支撑子域”等，例如“Delivery”是核心，“Customer”是通用，“Billing”是支撑——这个划分已经超出了软件开发人员的知识。

一家商场之所以能击败其他对手，原因未必是下单环节有什么不同，倒有可能是在配送环节下了大力气，或者支持的支付渠道多，或者客户服务环节抓得好。没有经过商业竞争的思考，武断地认为某个子领域是系统的“核心”是不合适的。

软件开发人员没有能力和责任做出商业竞争方面的判断。他要做的是，把涉众关于商业竞争方面的思考如实表达，并且用信息系统来封装其中适合封装的部分。

如果发现存在这样的“软件开发人员”，他有责任做出商业竞争方面的判断，那么说明这个人扮演了多个角色，既扮演涉众，也扮演软件开发人员。

【说明 2】

另一个可以选择的用词是“问题域 (Problem Domain)”，这也是之前大多数面向对象方法学的用语。但近年来，“问题域”和“解决方案域”等用语被领域驱动设计伪创新赶时髦胡乱使用，已经被严重污染。经过考虑，本书使用用语“核心域”和“非核心域”。

8.1.3 域之间映射和协作的套路

我们看一个人员管理系统的核心域类图，如图 8-2 所示。

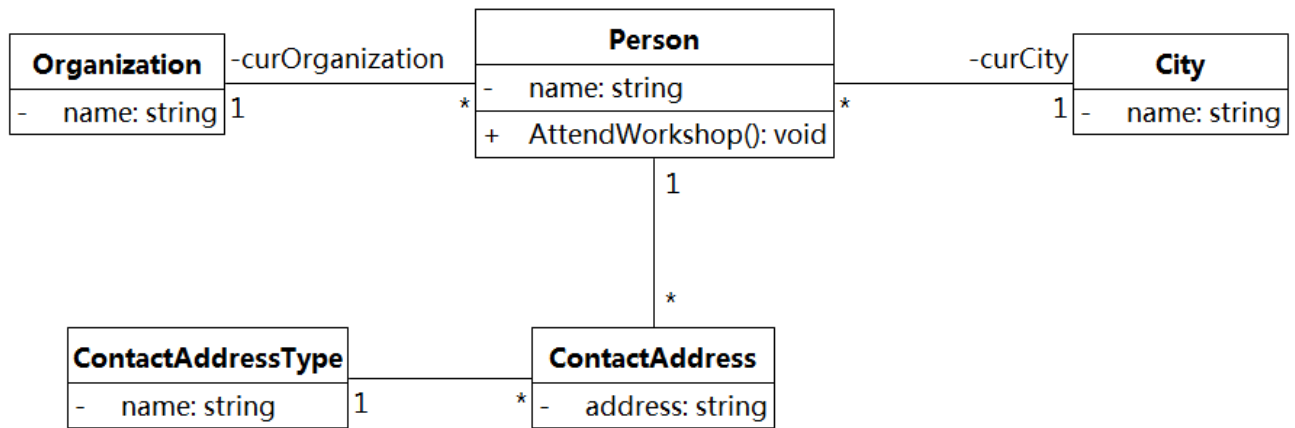


图 8-2 人员管理系统的核心域类图

如果将图 8-2 中的 Person 类映射为 C#实现，可能会得到图 8-3 的 C#代码：

```

16 public class Person {
17
18     private string name;
19     private Organization curOrganization;
20     private City curCity;
21     private List<ContactAddress> m_ContactAddress;
22
23     public Person(){
24
25     }
26
27     ~Person(){
28
29     }
30
31     public void AttendWorkshop(){
32
33     }
  
```

图 8-3 类的 C#实现（用 Enterprise Architect 映射）

如果将图 8-2 中的类映射到关系数据库，会得到图 8-4 所示的数据库结构：

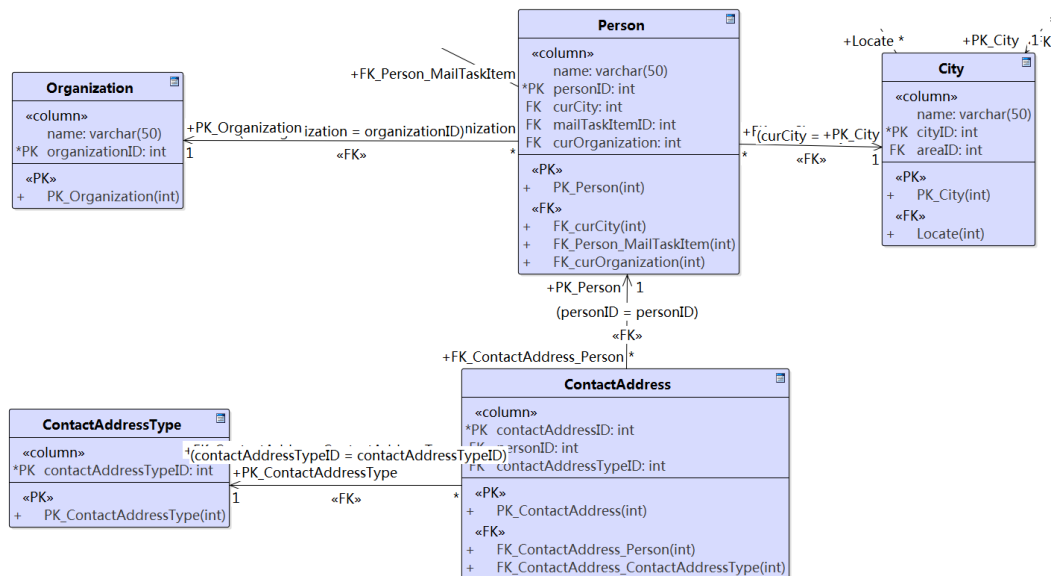


图 8-4 将类图映射到数据库模型(用 Enterprise Architect 映射)

如果采用某种对象 - 关系映射器框架（例如微软的 Entity Framework），Person 对象和数据库中的 Person 表里的一行可能会这样联系起来：

person1 = context.Persons.Find(ID)

注意，域和域之间的映射以及协作的套路，与域中的个体并不直接相关。

如果将以上内容中的 Person 改成 Dog，City 改成 Cat，映射的套路没有变化。即使我们调整了域之间的映射和协作的套路，得到的结果也会按照我们的调整有规律地变化，与域中的个体依然无关。

平时我们看到的一些“架构”，就是域之间映射和协作的一些套路。图 8-5 列出了现在常被提起的一些“架构”，可能在很多系统中都会观察到，即使这些系统的核心域及非核心域都有不同。

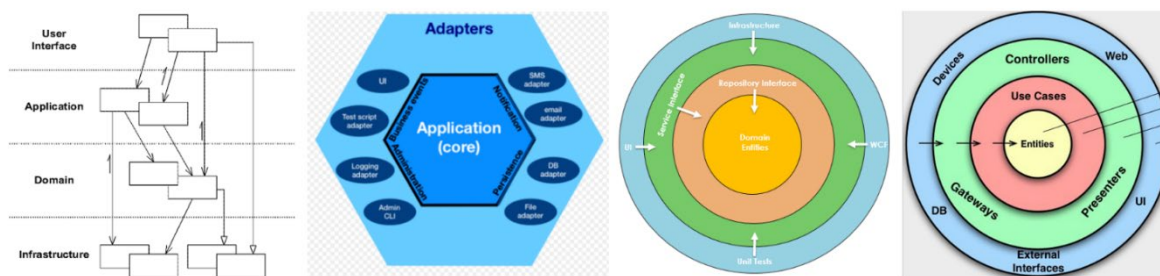


图 8-5 一些常见的“架构”

既然域之间的映射和协作有“套路”，过早地混合不同域的知识是不划算的。

如图 8-6 所示，假设三个域要考虑的知识点分别是 a 、 b 、 c 个，如果分开考虑，然后选择好域和域之间映射的套路，负担最小可以变成 $a+b+c$ ；如果混在一起考虑，大脑的负担最大会达到 $a \times b \times c$ 。只要 a 、 b 、 c 稍微大一些（都大于 $\sqrt{3}$ ），相乘的结果就会大于相加的结果，随着 a 、 b 、 c 进一步增大，差距会迅速增大。

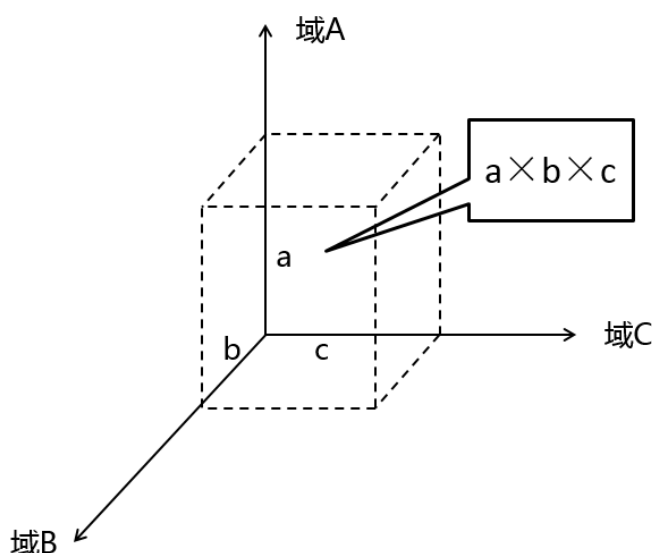


图 8-6 过早混合不同域的知识会增加大脑负担

过早地混合不同域的知识，会加重开发人员大脑的负担，导致开发

人员腾不出脑力来思考核心域中更深刻的问题,只好稍微折腾一下如图 8-5 的“域之间的架构”,心里安慰自己,我有“架构”了!却忘了,其实还没有触碰到最需要大脑去思考的核心域概念和逻辑。而这又很可能被巧妙地当成遮羞布——不是我不思考,而是要想的事情太多了顾不过来啊!

而这种微妙心态的进一步发展,会导致开发人员有意无意地混合不同域的知识,把复杂度弄成 $a \times b \times c$, 以此达成**废话刷工作量**——以最少的思考得到最多的“成果”。

近年最时髦的就是借 DDD 话语体系刷工作量了。例如,刚找出一个类 Order,然后周围就围上一圈 OrderFactory、OrderRepository、OrderService.....,洋洋得意地把工作量刷了好几倍。

我经常听软件组织的架构师向我介绍他们所开发系统的“架构”,口沫横飞,说的基本上都是图 8-5 的“域之间的架构”。好啊,真棒,我知道了。还有呢?没了?

构思那些“域之间的架构”是某些基础设施厂商或者方法学家的工作,我们挑一个适合自己项目的套路用上就行了。有什么问题,可以去请教用这个套路用得好的先行者或者与其合作。

“域内部的架构”,那些核心域概念和复杂逻辑,这是系统最值钱的地方。要是我们没有办法理清楚,别人是帮不到我们的。这才是大脑最该用的地方!

8.1.4 被忽视的核心域

8.1.4.1 鸟类里充兽，兽类里充鸟

当今的软件开发现实，核心域受到的重视远远不够。

我们经常会看到这样的场景：

开发人员张三喜欢研究“底层”。明明他的本职工作是用 C#编写生产管理系统，却不好好思考如何吃透核心域把项目做好，而是花时间研究编译器、操作系统甚至硬件。虽然工作是耽搁了，张三给人留下“勤奋好钻研”的印象。

甚至张三还可以写博客或公众号，在网络上成为媒体热爱的“网红程序员”。“网红程序员”很少探讨他当前所开发系统的复杂领域逻辑，更多是谈论某种语言或框架的特性。

张三这是在挑战高难度，勇攀科学高峰吗？非也。

脱离自己真正要去解决的问题，热衷于“钻研（其实只是学习）底层”，这样的行为更像是偷懒而不是勤奋。所谓“底层”也只是另一个领域的知识，那个领域自有另外的人去研究。玩票式的“钻研”，在真正专注研究这个领域的研究者看来，实在是不值一提。

但是人性的弱点如此，正如钱钟书所说：“蝙蝠碰见鸟就充作鸟，碰见兽就充作兽。人比蝙蝠就聪明多了。他会把蝙蝠的方法反过来施用：在鸟类里偏要充兽，表示脚踏实地；在兽类里偏要充鸟，表示高超出世。向武人卖弄风雅，向文人装作英雄；”。

比起研究真正需要自己研究的底层（例如质管部和生产部之间的冲突、生产过程中的各种概念之间的关系……），研究（其实只是学习）

MSIL 的语法等等，要容易和愉快得多。

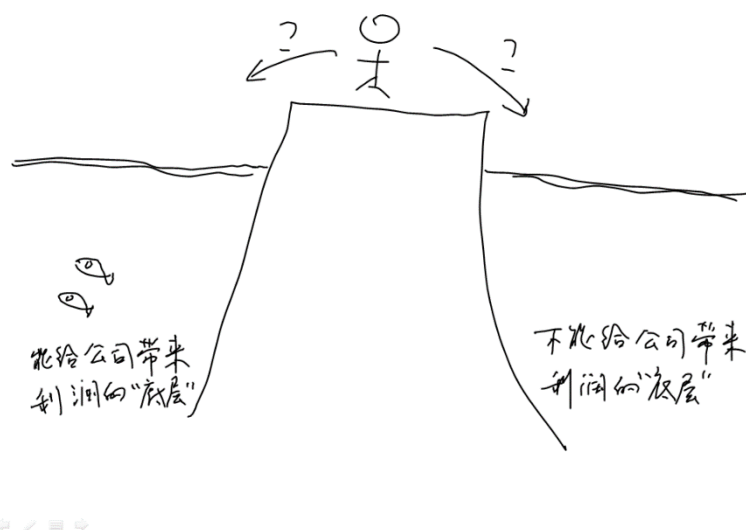


图 8-7 两个“底层”

8.1.4.2 “技术”、“底层”没什么特别

开发一个基础设施领域的系统，例如操作系统，只需要关注计算机的资源，不需要关注客户、订单、库存、病历等具体某个应用领域的概念。也就是说，“负载”比较低。

另外，基础设施领域有大量已出版教材和先行例子，高校也为计算机和软件相关专业学生开设了相应课程（Linus Torvalds 就是在大学教材中 MINIX 案例的激发下编写了 Linux）。这样，开发人员的大脑比较容易把握基础设施领域的复杂性。

在 2023 年 12 月 10 日用“操作系统”为关键字搜索当当网 (dangdang.com)，得到 315132 件商品，按 1/100 来挤水分也有 3000 多件。其中，一步步教读者如何自己编写操作系统的书也不在少数。

全部 > 图书 > 操作系统 共315132件商品					
分类	计算机/网络	教材	考试	工业技术	医学
出版社	人民邮电出版社	清华大学出版社	机械工业出版社	电子工业出版社	化学工业出版社
作者	鸟哥	戴有伟	刘遒	佚名	吴光科
品牌	异步图书	未读	新经典	机工华章	高等教育出版社
版本	人教版	人教PEP版	人教B版	统编版	
更多选项（适用学期、绘本馆、适用版本、年龄段等）					
综合排序 销量 ↓ 好评 ↓ 出版时间 ↓ 价格 ↓ ￥ - ￥ 配送至： 北京 ☐ 当当发货 ☐ 只看有货 ☐ 促销					



5折封顶

操作系统导论 美国知名操作系统教材 紧紧围绕操作系统的三大主题元素--虚拟化 并发和持久性

¥49.50 定价：¥99.00 (5折) **电子书：¥48.50**

[美] 雷姆兹 / 2019-06-01 / 人民邮电出版社

★★★★★ 8993条评论

[自营] [包邮] [券]

本书围绕虚拟化、并发和持久性这三个主要概念展开，介绍了所有现代系统的主要组件（包括磁盘和I/O子系统、文件系统）。全书共50章，分为3个部分，分别讲述虚拟化、并发和持久性。



图 8-8 当当网搜"操作系统"

市场已经对此做了回答。自己写（抄）一款操作系统并不难，但写一款能带来利润的操作系统太难了——这也同样适用于其他基础设施。

很多能够带来利润的应用系统，就没有基础设施那么好的待遇了。“自己动手写**应用系统”的书也不是没有，但其数量、深度和实用程度很难和基础设施的海量资料相比。

按道理，开发一个库存系统也应该可以不管基础设施，但遗憾的是，当前现实中大多数情况下还是要管。开发团队除了要懂得库存领域的知识，还需要懂 MySQL、Apache、Linux.....也就是说，“负载”比较高。

媒体的关注度也不够。一名上班族早上起来，先上厕所用智能马桶冲 PP，然后用智能电动牙刷刷牙，用微波炉热牛奶，边刷抖音边吃早餐，坐电梯下楼，开车去公司，用公司的业务系统工作。上面涉及到的七个智能系统中，估计只有抖音的开发人员可能会引起媒体的兴趣。

经常有“知名程序员”在谈到对建模（其实是 ABC 工作流）和 UML 的看法时，表示“对我来说不重要”，进一步探究会发现，这些“知名程序员”要么其所开发的系统是基础设施领域的系统，要么根本没做好自己的本职工作。不管是哪一种情况，这些“知名程序员”是知道谈论哪些内容更容易撩拨媒体从业人员的兴奋点的。

市场经济中，不存在哪个领域比其他领域更核心。如果像过去“以粮为纲”、“以钢为纲”一样，扭曲市场信号，硬性指定某个领域（芯片、操作系统）更高基，造就的多半是骗取纳税人金钱的投机分子。

8.1.5 要重视分析工作流

分析，就是从核心域的视角构思系统的内部机理。

在现在的很多软件组织中，分析工作流的技能被严重忽视。很多开

发人员上手就直接编码,原因并不是软件开发项目的核心域逻辑极其简单,不需要分析,或者他的大脑极其发达,在大脑里就可以完成分析,而是开发人员缺乏分析的技能,只好草草跳过这一步。

为了遮掩自己的无能,开发人员还会使用各种遮羞布——引入各种核心域逻辑之外的因素把水搅浑。

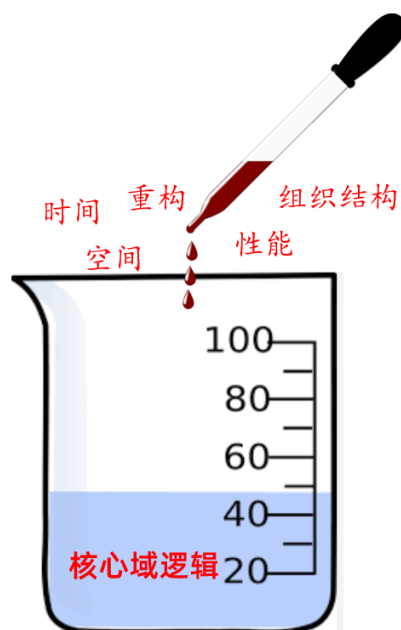


图 8-9 向核心域逻辑中引入其他因素

遮羞布一：时间

以“时间紧”、“敏捷”为借口,掩盖自己没有能力剖析复杂逻辑的事实——我是有能力剖析的,但时间太紧张了,等以后有时间吧!

遮羞布二：空间

借助“口头交流”、“白板”等容量小的介质,掩盖自己没有能力剖析复杂逻辑的事实——我是有能力剖析的,但白板空间太小了,只好简单画个“草图”了!

遮羞布三：功能需求之外的其他需求或设计因素

在思考核心域逻辑时，频频提出“这样会不会速度慢”（质量需求）、“我们想把它分成 N 个微服务，让不同团队用各自技术栈开发”（设计约束，也有可能是臆想的设计）等和核心域逻辑无关的因素，掩盖自己没有能力剖析复杂逻辑的事实——我是有能力剖析的，但还要考虑到这个因素、那个因素，所以精力就不够了。

遮羞布四：重构

以“后面再重构”为借口，结合其他遮羞布，掩盖自己没有能力剖析复杂逻辑的事实——我先随便写写，后面再重构，哎呀，没想到啊，时间来不及了（遮羞布一）。

关于重构，此处多说两句。

上 世 纪 80 年 代 末 ， Bill Opdyke (<http://laputan.org/pub/papers/opdyke-thesis.pdf>) 和 Bill Griswold (<https://cseweb.ucsd.edu/~wgg/Abstracts/gristhesis.pdf>) 等人归纳了一些调整代码结构的手法，称为“重构”，后经 Martin Fowler 等人推广而广为流传。

“重构”的知识可以看作是建模知识的一个子集。如果开发人员真的熟练掌握重构的手法，很多情况下他已经有能力直接建模系统的核心域逻辑得到更合理的结构，根本不需要先走很多弯路再回正路。

要是开发人员以“重构”为理由拒绝思考，很可能他的所谓“重构”

也是空话。

摸着石头过河是难免的，但应该在不得不摸的时候才摸，不应该假装看不见已有的路和桥，无论大小事都主动追求摸着石头过河。

当然，也有的人不是假装看不见路，而是真的看不见路——就是个睁眼瞎。不过，大脑不用思考，凭感觉摸着石头过河不停刷工作量，也是一种躺平的幸福。

用考试类比

学渣参加考试时，会这样遮掩自己的无能：

遮羞布一（时间）：抱怨时间紧张或者故意提前交卷——如果再给我一些时间，我肯定做得出来。

遮羞布二（空间）：故意带不好写的笔或抱怨草稿纸质量差、答题的地方太小——不是我不想好好答，可惜这个纸笔不给力。

遮羞布三（其他因素）：抱怨和学科知识无关的其他因素，例如要求用仿宋体答题——不是我不会，写仿宋体耗费了我很多精力。

遮羞布四（重构）：我先随便答，一会回来再检查。（结合遮羞布一）哎呀，来不及检查了。（结合遮羞布二）哎呀，答卷上地方不够了。

8.1.6 分析方法学历史的简单回顾

1958 年，John W. Young Jr.和 Henry K. Kent 发表 “Abstract formulation of data processing problems”，第一次提出在**独立于实现的抽象级别上**定义系统的规范。

AN ABSTRACT FORMULATION OF DATA PROCESSING PROBLEMS

by

John W. Young, Jr. and Henry K. Kent

The National Cash Register Company, Electronics Division
Hawthorne, California

SUMMARY

A data processing problem can be described in terms of four kinds of components:

- 1) Information set, P_i . This is a list of all possible information items belonging to the same class; from the items in these lists are drawn the data which flow through the system. Examples of items which make up information sets are customers' names and addresses, part numbers, unit prices, salesmen names, etc.
- 2) Documents, D_j . Documents are a collection of related information items D_{jk} . They are generally inputs or outputs, e.g., shipping notice, invoice, etc. Each document consists of K entries, e.g., date, customer's name and address, etc.
- 3) Relationships among the D_j , D_{jk} , and P_i . These will be discussed below.
- 4) Operational requirements of the system, e.g., maximum elapsed times.

图 8-10 摘自 “An abstract formulation of data processing problems” (Young JW, Kent HK, 1958)

1959 年, CODASYL (数据系统语言会议) 成立。1962 年, CODASYL 提出了一个和 Young/Kent 类似的模型, 称为 “信息代数” (Information Algebra)。

1970-1980 年代是结构化分析方法的时代, 主要贡献者有 Börje Langefors、Chris Gane、Trish Sarson、Tom DeMarco、Pin-Shan Chen、E. F. Codd 等人。结构化分析的主要建模方法是数据流图和实体-关系图, 这两者的结合, 让软件开发人员有能力剖析大型系统。

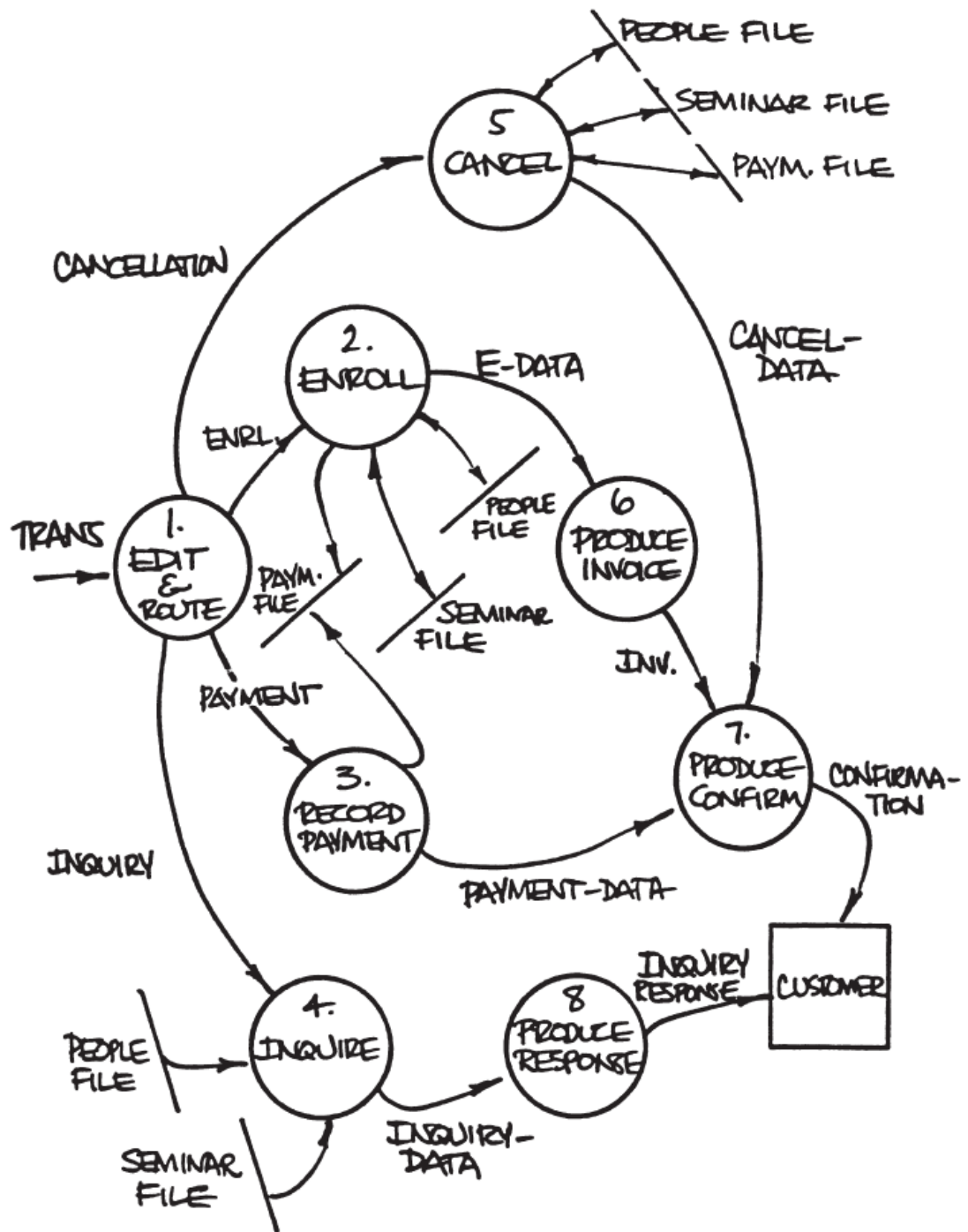


图 8-11 摘自 Structured analysis and system specification
(DeMarco T, 1979)

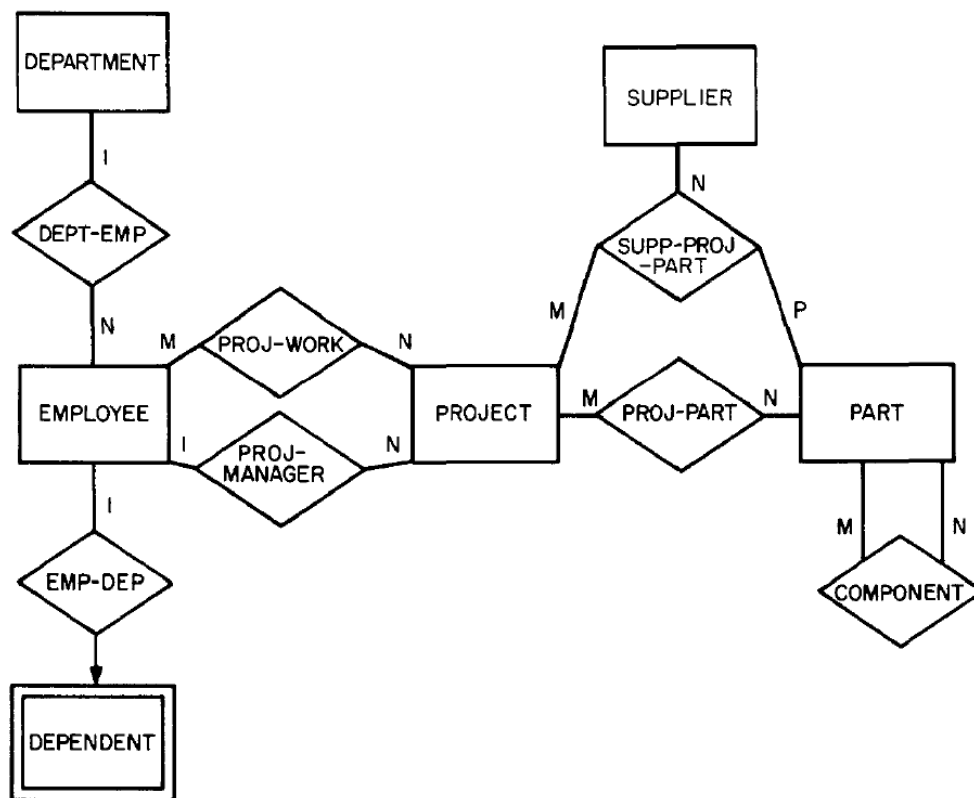
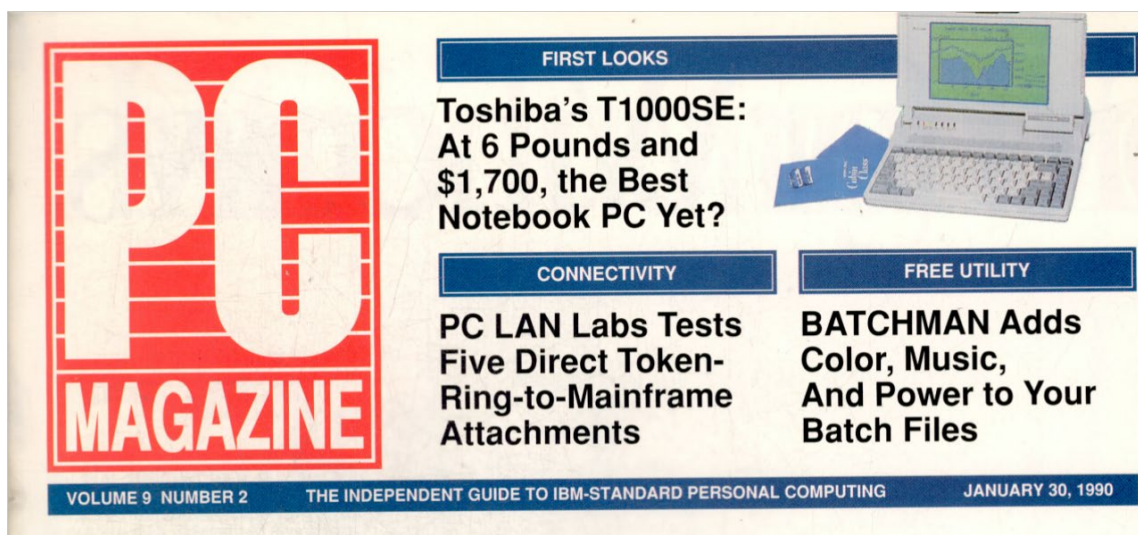


Fig. 11. An entity-relationship diagram for analysis of information in a manufacturing firm

ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976.

图 8-12 摘自 The Entity-Relationship model: Towards a unified view of data (Chen PPS, 1976)

1982 年, Nastec 公司开发出了 DesignAid, 这是第一款 CASE (计算机辅助软件工程) 工具。随后, 其他 CASE 工具陆续出现。据 PC Magazine 的 1990 年 1 月 30 刊统计, 当时已经有超过 100 家公司提供了将近 200 款 CASE 工具。



190	Features Table	effective ways to connect LANs to mainframes is to do it directly—through a token-ring to mainframe link that doesn't require a dedicated gateway. PC LAN Labs tests reveal which of five direct token-ring connections (from Attachmate, Banyan, DCA, IBM, and Novell) transfers data to and from a mainframe fastest.	signal the end of PostScript dominance and presage the dawn of a common OS/2-Apple font standard.
204	Performance Tests		
208	Editor's Choice		
213	SOFTWARE		
	CASE Tools for Designing Your Applications		
	Henry Fersko-Weiss/ Computer-aided software engineering packages can cost up to \$9,200—and still trim the software development life cycle enough to make them cost-effective. <i>PC Magazine</i> examines the capabilities of ten market leaders.		
220	Features Table		
251	Editor's Choice		
	has given PC-based CASE is an important step forward.		
	RIGHT FROM THE START		
	Design and analysis CASE tools are getting most of the attention today because automating the front end of the software development life cycle yields tremendous productivity gains at every subsequent stage. If the analysis and design of a program are done well, maintaining it is a snap. Today, IS departments spend about 70 percent of their time maintaining programs. Anything that makes the maintenance process easier frees up developers to create new systems.		
	Today, there are well over 100 companies offering nearly 200 different front-end CASE tools. In this article, <i>PC Magazine's</i> first look at CASE, we've honed in		
	don't sell their products as standalones. <i>PACBASE</i> from CGI Systems and <i>SDM/Standard</i> from AGS Management Systems are not included here because each requires a mainframe as well as a PC.		
	One product we chose to include, <i>POSE</i> , is too new to have a significant share of the market, but we think it's an exciting alternative worthy of consideration. We've also included a look at one other new product: Syscorp International's <i>MicroSTEP</i> , the first PC product to combine code generation with some of the design and analysis stages.		
	The packages that get full-fledged reviews here are <i>Analyst/Designer Toolkit</i> from Yourdon, <i>Auto-Mate Plus</i> from Learmonth & Burchett Management Systems, <i>DesignAid</i> from Nastec Corp., <i>Ex-celerator</i> from Index Technology,		
	of information through this system is indicated by arrows. Other techniques might use a similar diagram with somewhat different representations.		
	These differences can be merely stylistic or more substantial. In a Ward/Mellor transformation graph, you might have the exact same data-flow elements with two additions: dotted-line boxes with rounded corners to indicate control transformations—that is, changes in input from machines such as automated tellers or point-of-sale terminals—and dotted arrows to indicate the flow of machine inputs and outputs.		
	PICTURES TELL THE STORY		
	In the early stages of software development, creating diagrams of data flow, structure, and entity relationships requires		

图 8-13 摘自 PC Magazine 1990 年 1 月 30 日刊 (红框圈住的内容说明了工具的数量)

1980 年代后期, 面向对象的思想开始用于分析和设计。然后, UML

统一了表示法。这部分历史已经在本书第 1 章“UML 简史”部分讲述，此处不再赘述。

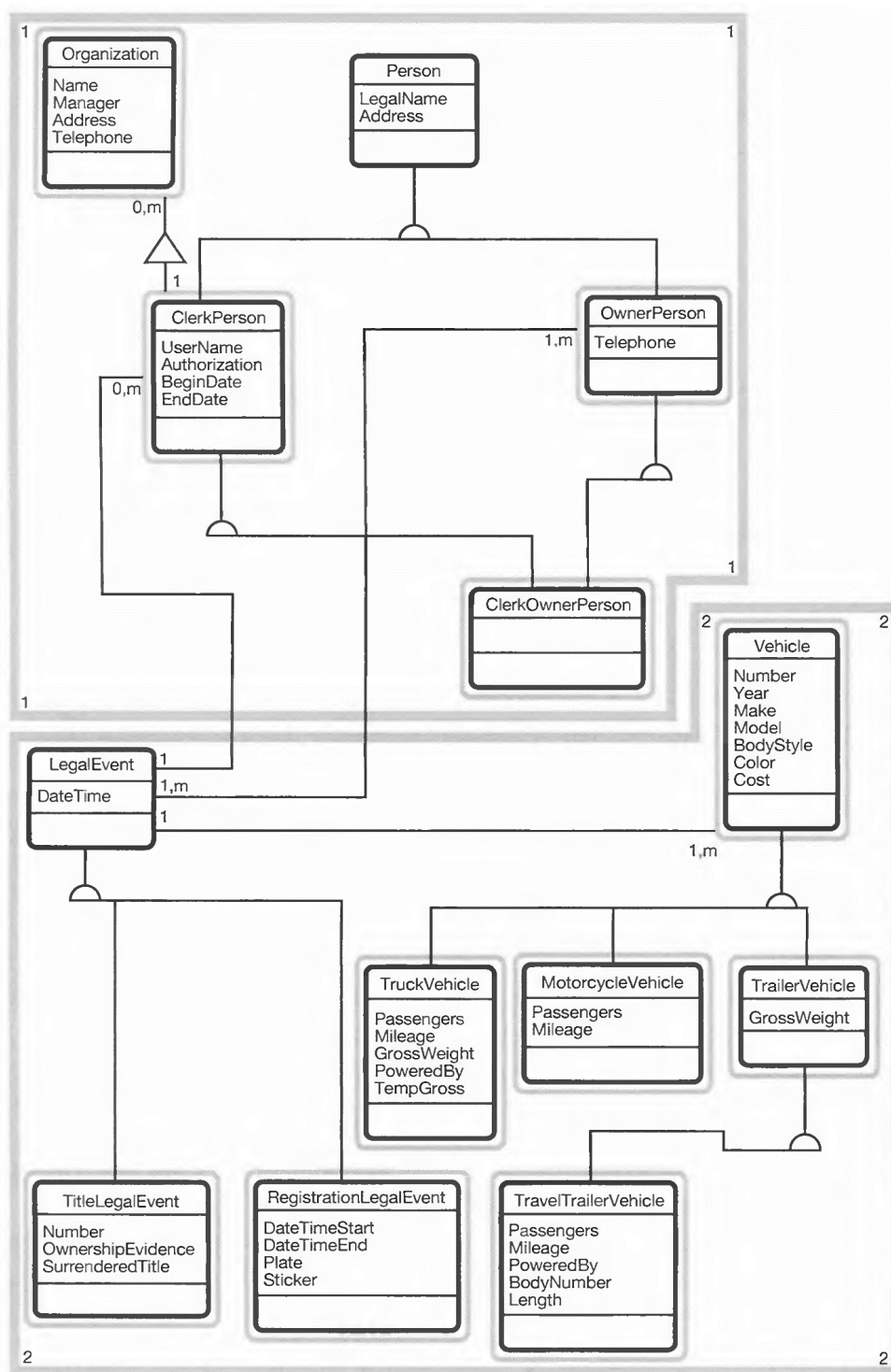


Figure 6.20: Registration and Title System—
Subject, Class-&-Object, Structure, and Attribute layers

图 8-14 摘自 Object Oriented Analysis, 2nd Edition (Coad P, Yourdon E, 1990)

Account

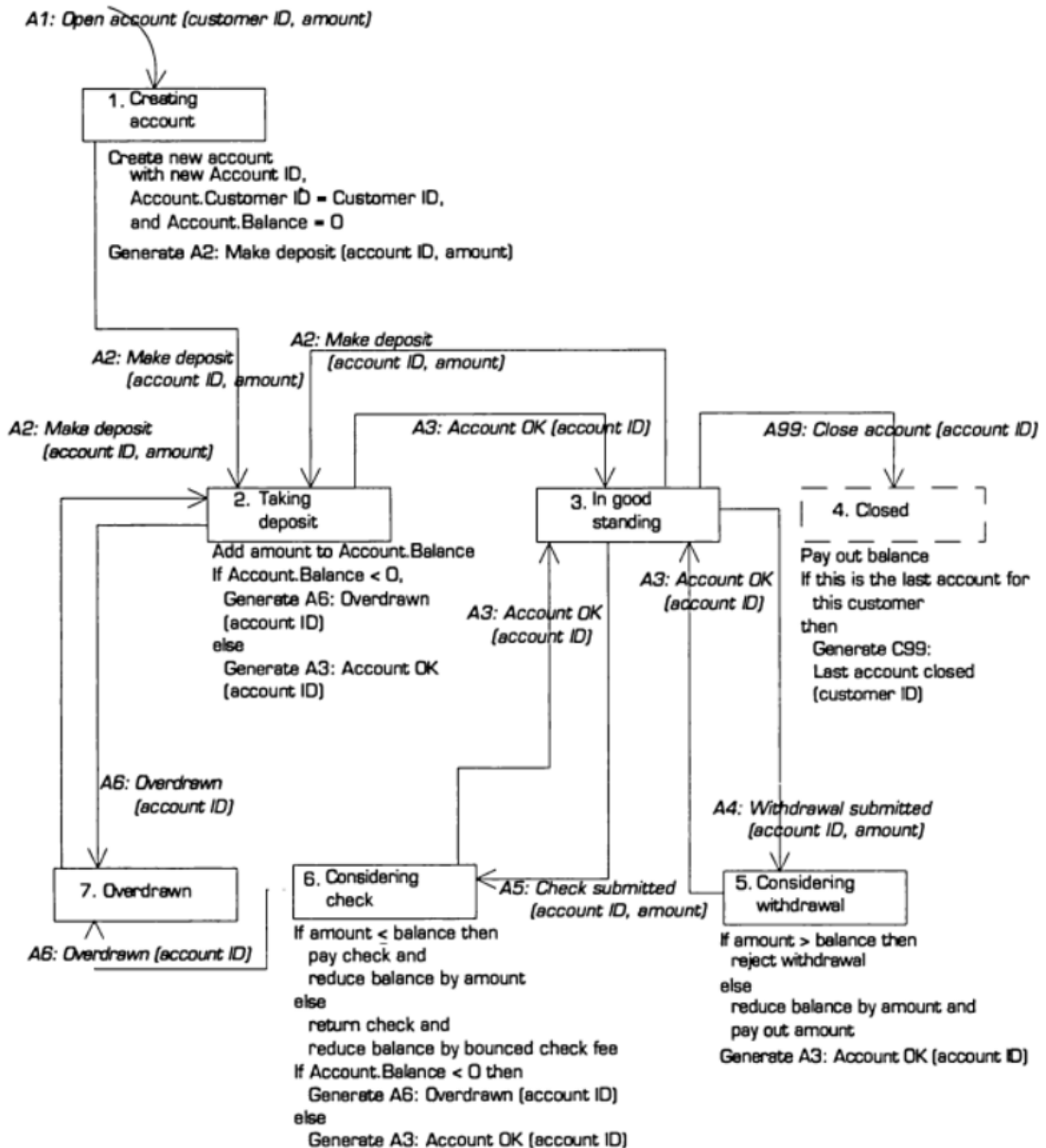


Figure 3.3.1: State model for the Account object

图 8-15 摘自 Object lifecycles. Modeling the world in states (Shlaer S, Mellor SJ, 1992)

8.1.7 本书使用的分析方法

分析模型描述系统要封装的核心域知识。

用什么建模概念来思考和描述核心域知识,可以有很多种选择。例如,“人”用不同的建模概念描述,可以说它是一个“类”,也可以说它是一个“类型”、一个“实体”。

本书使用面向对象的建模概念来描述分析模型,从三个视角来描述:

分析类模型:描述系统中各个类以及类的关系,如果用 UML 的类图表示,则为分析类图。

分析状态机模型:描述某个类的各个行为的逻辑,如果用 UML 的状态机图表示,则为分析状态机图。

分析交互模型:描述某些类在实现某个用例时的协作,如果用 UML 的序列图表示,则为分析序列图。

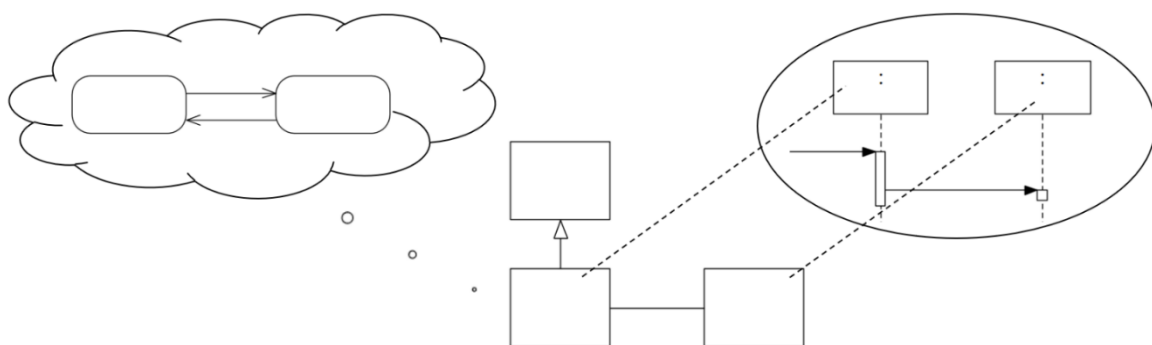


图 8-16 本书的分析方法所使用的 UML 图形

需要说明的是,虽然我们用的是面向对象的分析方法,也就是说,用面向对象的概念来剖析核心域知识,但不意味着你的系统一定要用特定的“面向对象”编程语言、特定的存储方式或物理分布形式来实现。

也许你使用的编程语言是面向过程语言,例如 C;也许你使用的编程语言是函数式语言,例如 F#;也许你使用的存储系统是关系数据库

系统, 例如 SQL Server; 也许你使用的存储系统是非关系数据库系统, 例如 MongoDB; 也许你的系统运行在同一台机器上, 也许是分布在很多台机器上.....

不管你的系统的实现方式和运行形态如何, 从分析过渡到设计时, **变化的只是分析到设计的映射套路**。如果设计所使用的非核心域比较“面向对象”, 那么映射套路会比较直观一些, 否则, 就需要一定的转换。但无论如何, 如前文所说, **这个套路和具体的核心域知识没有关系**, 我们并不需要针对每一个核心域概念逐一花费脑力去思考它。

我们之所以选择在分析工作流使用面向对象的分析方法, 是因为从思考深度和表示的严谨程度来看, 面向对象的分析方法以及 UML 表示法目前仍然是剖析和整理核心域逻辑的最佳选择。

本书在设计工作流的内容, 会展示分析模型和各种实现方式的映射套路。

8.1.8 自测题



扫 码 或 访 问

http://www.umlchina.com/book/quiz08_01.html 完成在线测试, 做到全对以获得答案。



1[多选]

关于分析和设计的区别，以下说法不恰当的有：

- A) 分析着眼于“系统做什么”，设计着眼于“系统怎么做”。
- B) 分析和设计分离的好处是，先全局思考整个系统的各个类以及类的关系，再有规律地映射到实现的平台和语言，这样就减少了反复试错的成本。
- C) 有时候，在分析工作流也会考虑内存、网络带宽等概念。
- D) 分析注重业务，设计注重技术。

2[单选]

掌握 MVC、MVP、MVVM、六边形、洋葱型.....等模式或架构，并不能解决分析的问题，原因是：

- A) 它们描述的是域之间的协作。
- B) 它们没有得到广泛使用。
- C) 它们没有体现面向对象的思想。
- D) 它们不够敏捷。

3[单选]

第一款 CASE（计算机辅助软件工程）工具是：

- A) Rose
- B) ERwin

C) FlowChart/360

D) DesignAid

4[多选]

本章中目前为止提到的在分析时逃避思考遮羞布有以下哪些？

A) 不使用 UML 的类图 and 状态机图。

B) 谈论性能问题。

C) 抱怨免费或便宜的建模工具太弱，要求购买更贵的建模工具。

D) 引入微服务架构以简化核心域逻辑。

5[多选]

以下说法正确的有：

A) 分析模型就是用核心域术语表达内容的模型。

B) 分析模型概要地描述核心域知识，设计模型将核心域知识细化。

C) 面向对象的分析模型不妨碍使用面向过程的设计。

D) 口头表达也可以表达分析模型。

6[单选]

有一篇文章，作者在白板上画了一个类图，然后开始掰着指头数这个类图缺什么，“没考虑到持久化”，“没考虑到对象的创建”.....然后得出结论：画这个类图不如直接编码。根据本节的知识，以下正确的说法是：

- A) 作者不了解核心域和非核心域分离的重要。
- B) 别急,这个图会越来越细,逐渐添加作者认为缺少的那些东西。
- C) Talk is cheap. Show me the code.
- D) 敏捷是建模的精髓,加上这些就不敏捷了。

7[多选]

以下系统中, 适合用面向对象的分析方法建模核心域逻辑的有:

- A) 火箭发射控制系统
- B) 函数式语言的编译器
- C) 视频剪辑软件
- D) 关系数据库管理系统

8[多选]

以下系统中, 适合用面向对象的分析方法建模核心域逻辑的有:

- A) 用 SQL 存储过程来实现核心域逻辑的企业应用
- B) 用 C 语言实现核心域逻辑的嵌入式应用
- C) 用 JavaScript 实现核心域逻辑、读取本地文件, 不需要服务器的小游戏
- D) 在微软 Azure 上运行的企业应用

番号	名称	时长	价格（元）	备注
SYS-002	住宅安全系统	7小时14分	469	SysML+EA
SYS-001	水蒸馏器	9小时04分	619	SysML+EA
QH-001	《软件方法》强化自测题解析第1集	3小时18分	299	总纲（一）、业务建模（一）、需求（一）、分析（一） 最好先上过前置课程GJ-001或GJ-002
JD-001	《软件方法（上）》题目解答解析和深入展开	7小时24分	479	可结合《软件方法》书或者UML-001使用。
GJ-001	业务建模和需求高阶	9小时48分	849	最好先上过前置课程UML-001
GJ-002	分析和设计高阶	10小时38分	999	最好先上过前置课程UML-001
UML-001	软件需求和设计方法学全程知识讲解	7小时07分	129（特价）	配合EA-***和ST-***番号使用 2年内可免费更新
EA-030	制药病毒培育记录	5小时15分	419	纯实作，结合知识讲解UML-001使用
EA-029	石油钻井管理平台	5小时13分	419	纯实作，结合知识讲解UML-001使用
EA-028	高压注射器	4小时08分	399	纯实作，结合知识讲解UML-001使用
EA-027	智慧公寓系统（配可执行代码）	5小时43分	469	纯实作，结合知识讲解UML-001使用
EA-025	伺服驱动器	3小时59分	389	纯实作，结合知识讲解UML-001使用
EA-024	高校毕业生就业系统	6小时33分	469	纯实作，结合知识讲解UML-001使用
EA-023	粮食拍卖系统	6小时6分	459	纯实作，结合知识讲解UML-001使用
EA-021	预约住院系统（配可执行代码）	5小时34分	449	纯实作，结合知识讲解UML-001使用
EA-018	司法调解平台（配可执行代码）	4小时57分	429	纯实作，结合知识讲解UML-001使用
EA-017	信贷信息系统（房产抵押评估）（配可执行代码）	6小时30分	499	纯实作，结合知识讲解UML-001使用
EA-016	博物馆安全系统	5小时10分	419	纯实作，结合知识讲解UML-001使用
EA-015	跨组织结算系统	5小时27分	429	纯实作，结合知识讲解UML-001使用
EA-014	患者远程求医系统	4小时23分	409	纯实作，结合知识讲解UML-001使用
EA-013	期货交易仓单系统	5小时11分	419	纯实作，结合知识讲解UML-001使用
EA-012	市场部营销活动系统（配可执行代码）	5小时57分	459	纯实作，结合知识讲解UML-001使用
EA-011	停车管理系统	4小时22分	409	纯实作，结合知识讲解UML-001使用
EA-010	财务软件	5小时47分	439	纯实作，结合知识讲解UML-001使用
EA-009	设备维护知识图谱系统	5小时23分	429	纯实作，结合知识讲解UML-001使用
EA-008	合同管理系统（催款）	3小时07分	389	纯实作，结合知识讲解UML-001使用
EA-007	拼多多（拼单购物系统）	2小时47分	379	纯实作，结合知识讲解UML-001使用
EA-006	迪迪出行（网约车系统）	2小时56分	389	纯实作，结合知识讲解UML-001使用
EA-005	微信餐馆（配可执行代码）	3小时25分	199（特价）	纯实作，结合知识讲解UML-001使用
EA-004	会议室管理系统（配可执行代码）	2小时29分	369	纯实作，结合知识讲解UML-001使用
EA-003	房产中介考勤系统	2小时08分	349	纯实作，结合知识讲解UML-001使用
EA-002	制造执行系统MES	2小时54分	449	纯实作，结合知识讲解UML-001使用
EA-001	三方采购平台	2小时01分	339	纯实作，结合知识讲解UML-001使用
ST-001	制造执行系统MES	2小时50分	149（特价）	工具为StarUML 纯实作，结合知识讲解UML-001使用

建模示范视频：<http://umlchina.com/url/video.html>

8.2 建模步骤 C-1 识别类和属性

8.2.1 面向对象

面向对象就是一些假设。

如果不认可“面向对象”的假设，也可以分析系统的核心域知识，只不过用的方法不叫“面向对象方法”，叫“面向过程”、“面向组件”、“面向肥皂”、“面向武德”都行，看你的假设是什么了。

面向对象的思考方式比目前的其他思考方式要好一点，原因不是计算机喜欢面向对象或者面向对象更接近于计算机的底层，而是面向对象的思考方式更能帮助人脑去剖析复杂问题。如果计算机有感情，估计它应该更“喜欢”人类用机器语言直接给它发指令，因为这样自己就不用受累搞什么编译、链接。

面向对象更能帮助剖析复杂问题，不意味着面向对象的思考方式比其他的思考方式更容易掌握，而且随着你掌握了更强有力的思考工具，更复杂的问题就会扑面而来。这些问题早已存在，只不过之前你没有能力来发现和对付它们——“古人很少死于癌症”。

正如第 1 章在批评伪创新时所提到的，三角函数更能解决复杂问题，不意味着它比全等三角形、相似三角形更容易掌握。

伪创新经常把“简洁”和“容易学”混淆，动不动嚷嚷“大道至简”，其实背后意思是“不动脑子也能学会的才是好方法”。

像下面这些足够“简洁”而且可以解决之前困扰人们的一大片难题，

但并不“容易学”。

线性方程组有解 $\leftrightarrow$ 系数矩阵与增广矩阵有相同的秩

方程存在根式解 $\leftrightarrow$ 方程的群存在因子全为素数的子群系

当使用面向对象的方法来分析系统时，我们引入的**第一个假设**是：

系统由"对象"这样一种东西构成，对象封装了数据和行为。

严格来说，我们建模的是类（也许叫“基于类的方法”更合适），即对象的“模板”，而不是对象。对象是运行时（或模拟运行时）才产生的。

我们通过抽象思维把具有共同特征的对象集合归纳为"类"，对象看作类的实例。归类是人类认知的一种基本技能，其哲学讨论可以追溯到柏拉图的理型论（Theory of Forms）。

我们引入的**第二个假设**是：

对象在一个"对象空间"中运行，在这个空间中发生的所有事情消耗的时间为零。

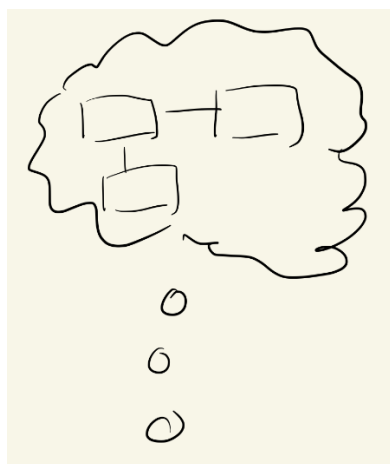


图 8-17 想象的"对象空间"

您可以认为这个"对象空间"存在于大脑中,也可以把"对象空间"想象成一台存储空间无限大,通信和运算速度无限快且分布在全宇宙的超级计算机。在这个假设下,不用考虑什么硬盘、内存、Cache、加载,只需要聚焦于思考核心域知识。

当然,“时间为零”、“无限快”是不可能的。我们的宇宙,目前因果关系的最快速度是光速。

当前现实中的计算机和网络,要发生一段对人类有意义的因果关系,例如,从提交关键词到返回查询结果,时间估计会以秒来计,不过,作为人类的涉众可能对此已经表示满意了。

如果当前的计算机和网络资源能够满足人们对性能的要求,那么设计模型(代码、存储.....)和分析模型之间映射会非常直接。

反之,如果出现不可调和的性能问题,设计模型可能会有所调整,例如,添加一些冗余,但这样的调整和具体的核心域知识无关,可以把

它们归纳成一些套路，出现相应问题时按照套路调整即可，不需要在分析时考虑这些问题。

“不考虑性能”这一点，可以用来判断你思考的问题是分析问题还是设计问题。

我们可以针对分析模型里的元素，一个一个问，“如果没有它，会怎么样”，如果回答是“会有性能问题”，那么，可以从分析模型中把它删掉。

例如，类图中有一个冗余的类，问“如果没有它，会怎么样”，答“查询可能会慢”——可以删掉。状态机图里有一个状态“Transient”，问“如果没有它，会怎么样”，答“会漏掉某些数据没有持久化”——可以删掉。

但如果回答是“没有它，系统就没法履行自己的责任了，因为要做的系统就是一个持久化框架”，那就不一样了。

分析模型受到设计的污染，很容易导致批量的废话刷工作量，导致没有时间思考应该重点思考的核心域逻辑。**当然，正如前文所说，这也可能正是某些人故意寻找的遮羞布。**

8.2.2 三种分析类

8.2.2.1 Ivar Jacoson 的假设

我们再引入一个假设：

系统中存在三种分析类：边界类（Boundary Class）、控制类

(Control Class) 和实体类 (Entity Class)。

这个假设借用了 Ivar Jacobson 在 “Object-Oriented Software Engineering: A Use Case Driven Approach” (Jacobson 1992) 中的思想。其他面向对象方法学可能并不认同这个假设，也就是说，面向对象的分析不一定需要这个假设。

在 UML 模型中，我们可以用 Ivar Jacobson 建议的构造型 (Stereotype) 来表示三种分析类，如图 8-18。

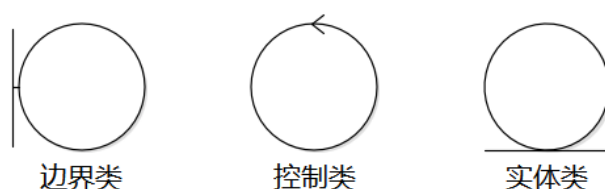


图 8-18 三种分析类的构造型

一些 UML 工具 (如 Enterprise Architect、Visual Paradigm) 已经内置了这些分析类构造型。如果使用的建模工具没有内置这些构造型，可以自己添加如 “<<边界>>” 等文字构造型；或者不用构造型区分，通过给类起名 “某某接口”，“某某控制”，也有助于了解该类在系统中扮演的角色。这一点，和第 3 章讲到业务工人、业务实体时的做法是一样的。

在设计 workflow，三种分析类可以映射到任何实现架构，包括但不限于 MVC、MVP、MVVM、六边形、洋葱型……甚至映射到不做任何分割的 “架构”。

图 8-19 列出了三种分析类的责任、和需求的关系以及命名。

构造型	责任	和需求的关系	命名
边界类	输入、输出以及简单的过滤	每个有接口的外系统映射一个边界类。	外系统名称+接口
控制类	控制用例流，为实体类分配责任。	每个用例映射一个控制类。	用例名称+控制
实体类	系统的核心，封装核心域逻辑和数据。	用例和实体类的关系是多对多的，一个用例可以由一到多个实体类协作实现，一个实体类可以参与一到多个用例的实现。	领域概念名称

图 8-19 分析类的责任、和需求的关系以及命名

图 8-20 用序列图展示了三种分析类之间的协作。

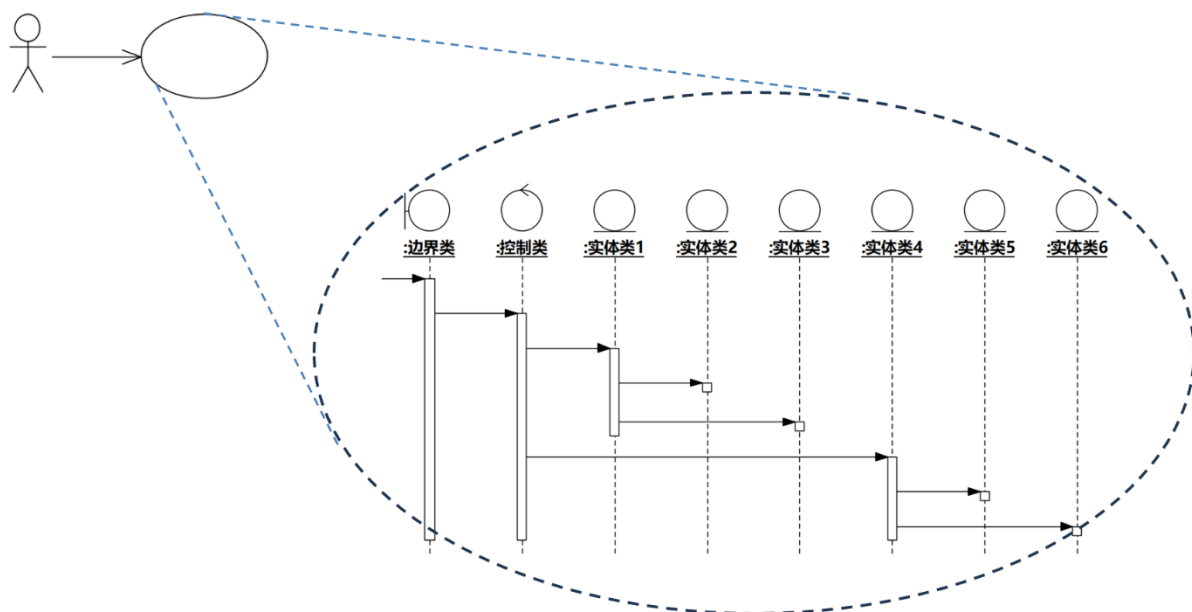


图 8-20 三种分析类在系统中的协作

执行者先把消息发给边界类对象。边界类对象履行它有能力履行的责任，然后把它没有能力履行的责任委托给控制类对象。控制类对象就像总裁办，不做具体工作，只是将责任分解后分配给实体类对象。

分配给实体类对象时，如果某个对象被其他对象组合，应该先分配给组合它的对象，再由该对象分配给它。DDD 话语体系中的“聚合 (Aggregate)”和这一点类似，本书在后文讲述类关系的章节中会进一步阐述其中差别以及“聚合 (Aggregate)”的伪创新。

最后，由边界类对象向外系统反馈信息，完成一个交互回合。

8.2.2.2 关于边界类

边界类的责任是接受输入、提供输出以及做简单的过滤。

图 8-19 中提到边界类的映射方法——每个有接口的外系统映射一个边界类。这里说的“有接口的外系统”不仅包括系统执行者，还包括

仅接受系统输出信息的外系统。

以 2018 版《软件方法（上）》UMLChina 系统案例中的"时间→发送公开课通知"用例为例。该用例进行过程中，系统会向软件开发人员发送公开课通知，同时还要向 UMLChina 助理反馈发送通知的进展。软件开发人员和 UMLChina 助理在这个用例中仅仅是接受输出，没有输入信息给系统,但系统可以分别设置一个边界类来封装向软件开发人员和 UMLChina 助理反馈信息，如图 8-21 所示。

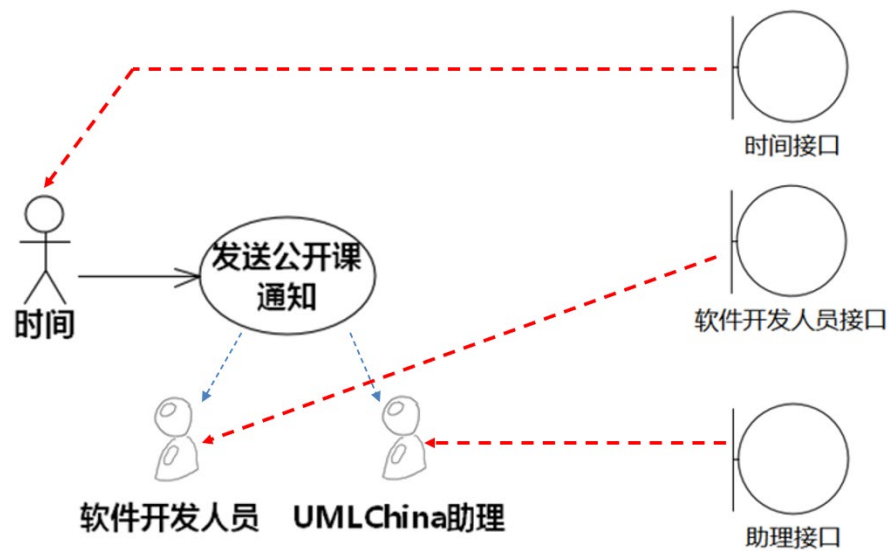


图 8-21 外系统映射边界类

图 8-21 中，“时间”映射一个“时间接口”，“软件开发人员”映射一个“软件开发人员接口”，“助理”映射一个“助理接口”。

外系统如果是人，对应的边界类也可以叫“**界面”，例如“助理界面”。本书就不区分了，一律起名“**接口”。

分析工作流的边界类不暗示任何实现方案。在总责任相等的前提下，它和实现的映射是多样的，可以是图形、文本、语音、远程调用.....

即使使用图形界面实现,也不能简单认为一个边界类对应一个窗体(Form)。一个边界类的责任可以拆解到多个窗体上,一个窗体也可以和多个外系统交互。如何组织这些责任,应该从外系统的角度来考虑,而不是从用例或实体类的角度来考虑。

图 8-22 中,“助理接口”边界类被圈住的几个责任来自不同用例的步骤,但在使用图形界面实现时,可以放在面向助理的、通知专用的同一个窗体中。

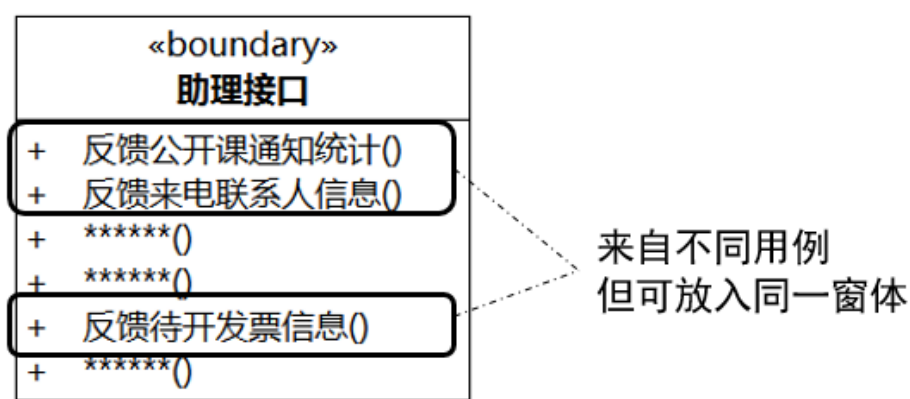


图 8-22 边界类责任的组织

类似的例子还有:一份申请,需要通过系统审批三次,也就是三个不同的用例。在图形界面实现中,可能不需要准备三个窗体,部门主管、财务、副总三个审批人可以在同一窗体上工作,但部门主管、财务、副总各自有对应的分析边界类。

如果某个外系统和系统的交互很多,对应边界类的责任可能会有很多。有的做法推荐按“外系统+用例”的组合映射边界类,这样可以减少一个边界类上的操作个数。本书不推荐这样做,因为这已经隐含着先入为主“按用例划分边界”的意思,不利于最后得到合理的边界。

尽量保持一个外系统映射一个分析边界类,如果操作很多,可以将

从外系统角度观察可能要分在一组的操作移到一起，EA 等工具可以随意定制属性和操作的上下显示顺序。

需要提醒的是，外系统映射的只是边界类，并不映射实体类。在外系统是人的时候，经常会有人犯这样的错误。例如以下用例规约片段：

1. 助理选择公开课，请求创建通知任务
2. 系统验证所选公开课适合创建通知任务

“助理”是执行者，映射一个边界类“助理接口”是可以的，但如果映射一个“助理”类，那就错了，如图 8-23。

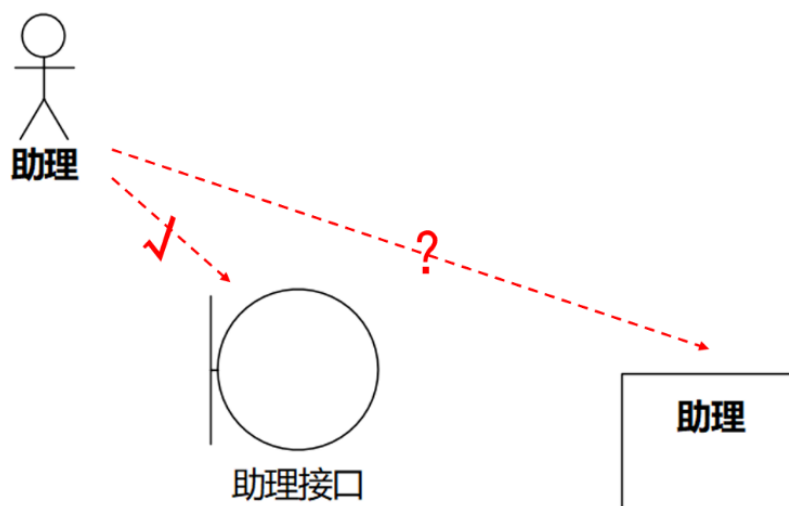


图 8-23 外系统不映射实体类

系统是否需要一个“助理”类，要看系统是否需要维护助理的信息。如果需要，会在某个用例规约的某个地方体现，例如，可能会有一个步骤：

7. 系统保存通知任务

绑定一个字段列表：

7. 通知任务=4+创建时间+创建人

这个“创建人”就是助理，说明系统需要记住助理的信息，这时才会有“助理”类。

但并不是所有的系统都需要保留人的信息。例如，乘客坐电梯上楼，乘客是电梯系统的执行者，但电梯系统可能不需要“乘客”实体类，因为它不需要记住乘客的信息。

当然，有朝一日，电梯升级为防疫电梯，用例规约里有：

4 乘客提供身份标识

5 系统验证身份标识合法

6 系统记录乘客信息和入厢时间

这时，电梯系统里就有“乘客”实体类了，因为系统要记住乘客的信息。

当然，虽然电梯系统没有“乘客”类，但会有“乘客接口”类，可能的类图和常见的实现方式如图 8-24。



图 8-24 “乘客接口”及其常见的实现

8.2.2.3 关于控制类

控制类相当于用例在系统中的“代理”，它的责任是控制用例流，为实体类分配责任。如果在分配责任时发现控制类只起到传递的作用，没有起到分解和分配的作用，也可以把控制类去掉。

因为每个用例直接映射一个控制类，可以用“用例名称+控制”来为控制类命名。

因为构造型已经包含“边界类”、“控制类”的概念，严格来说，“员工接口”、“审批控制”类名后面的“接口”、“控制”可以省掉，在分析映射设计时，再通过映射套路把它补上去。不过，考虑到可能还会有“员工”、“审批”实体类，而且很可能会一起出现在同一张序列图上，仍然保留“接口”、“控制”的尾巴看起来更顺眼一些。

8.2.2.4 关于实体类

边界类与外系统、控制类与用例的映射关系很明显，所以识别边界类和控制类不需要思考，直接按照上面的套路映射即可，甚至可以先不映射，推迟到画分析序列图时再加上去。

有的分析方法学如 ICONIX 提倡一种 Robustness Diagram，认为可以通过它来帮助寻找类。开发人员一用确实感觉很舒服，噼里啪啦就发现好多类，有一种“我已经取得了不小成绩”的错觉，不过要是仔细看看，就知道“发现”的大多是边界类、控制类。这些类其实用不着刻意去发现，只要按照图 8-19 的套路映射即可。

最难的工作——寻找实体类以及它们之间的协作，Robustness

Diagram 却是寥寥带过，甚至容易误导建模人员把实体类和用例一一对应。所以，本书不推荐开发人员额外花时间画 Robustness Diagram。

★和前文多次提到的一样，凡是不需要思考就可以得到很多“成果”的“方法”，都容易成为懒人摸鱼的遮羞布。

建模人员的思考工作量应该花在识别实体类上。一个用例需要哪些实体类协作实现、如何协作，一个实体类会参与哪些用例的实现，这是一个多对多的映射，需要由建模人员的大脑决定哪种映射最好。

因此，本章以下内容提到的“类”，缺省意思为“实体类”。

边界类、控制类的命名有“**接口”、“**控制”的尾巴，实体类的命名就不要尾巴了，直接用核心域概念命名即可。例如，命名为“员工”而不是“员工实体”。

8.2.2.5 不存在“系统”这个类

“系统”的概念是需求工作流的概念。在需求工作流，我们把系统看作一个对外提供服务的整体。在分析工作流，“系统”的概念已经被打碎成很多个类，“系统”这个词不需要识别成类。

图 8-25 表达了不同工作流视角下的目标系统。

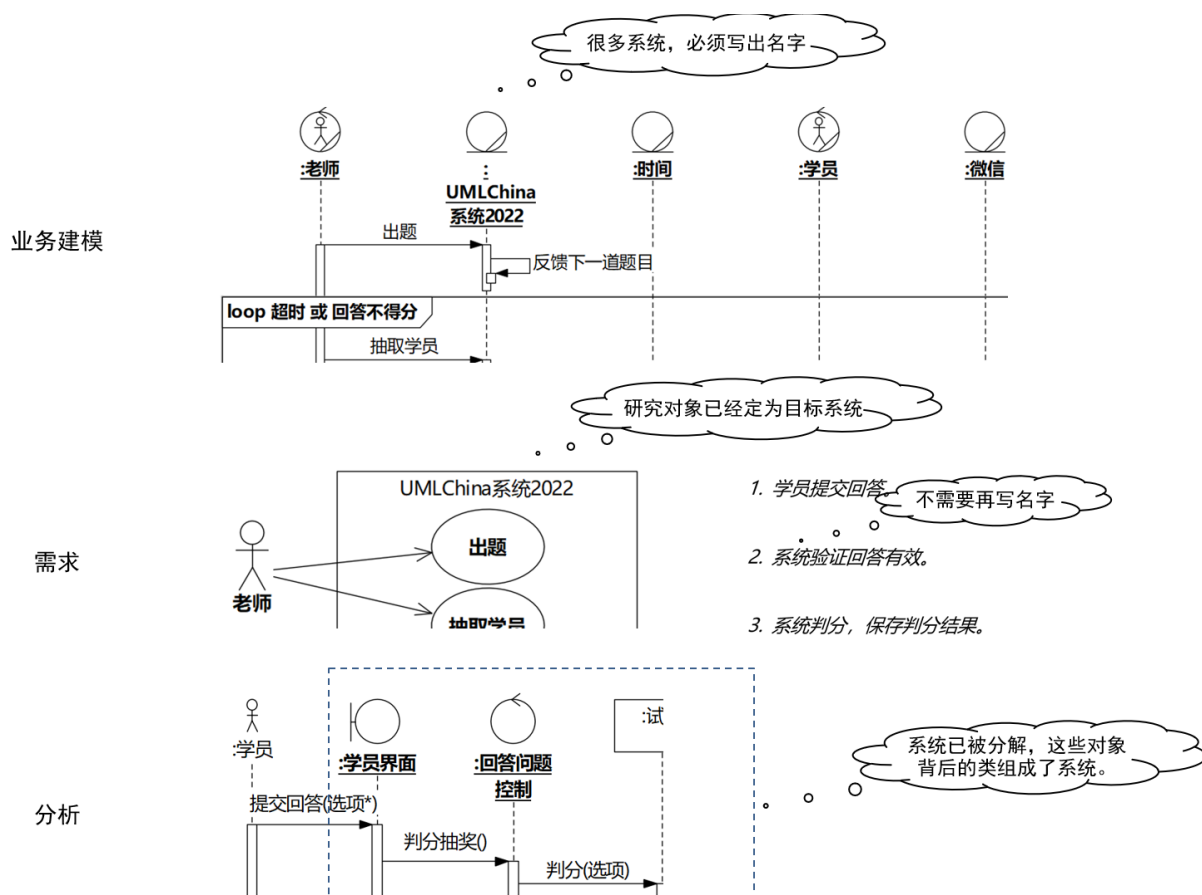


图 8-25 各 workflow 如何称呼目标系统

在业务建模 workflow，研究范围是组织，而组织中有很多系统，在业务序列图上提到目标系统时，只是说“系统”二字无法让人理解指的是哪一个系统，需要写出目标系统的名字“***系统”。

在需求 workflow，研究范围是目标系统整体，此时，聚光灯已经打在目标系统身上，不需要再写目标系统的名字，写“系统”二字即可。

就像公司开表彰会，老总宣布优秀员工名单时，要说名字“罗永昊”、“罗阵宇”，轮到优秀员工罗永昊上台发言时，罗永昊称呼自己就不好再说“罗永昊”，说“我”、“在下”、“小弟”就行了。

*用自己的名字以及第三人称来称呼自己，往往代表一种极度的“自信”。例如：

“凯撒注意到这事，把他的军队撤到最近的一座山上去”（《高卢战记》）

“老胡觉得这件事实在不应该”

“婷婷想吃冰淇淋嘛” *

而在分析工作流，研究范围是系统的内部构成。系统已经被分解成很多个类，这时就不能再说“系统”了。

有的建模人员会把“系统”识别成一个类，画成这样：

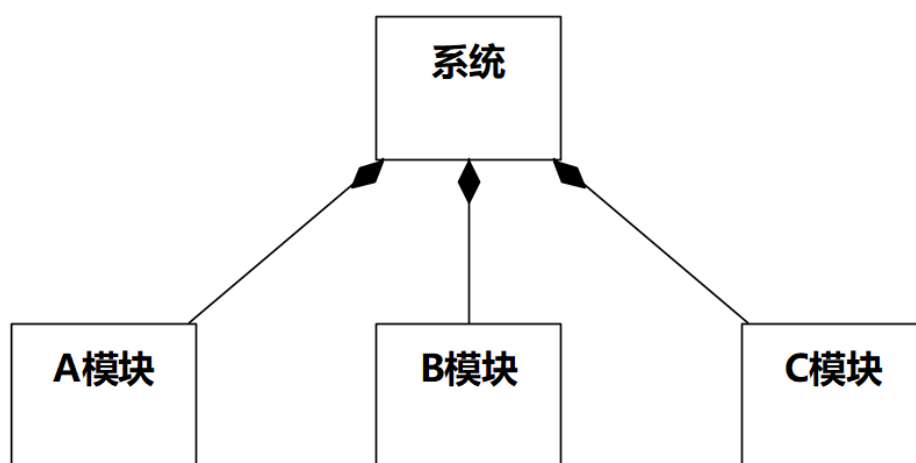


图 8-26 无意义的类图

这种图画出来既没有根据（你怎么知道是这样分解？）又含义模糊（模块指的是需求包还是组件？），对剖析系统的复杂性没有帮助，但给建模人员带来一种**虚假的成就感**：我描述了几个类以及类的关系，而且还是组合关联（还可以联想到高大上的“组合优于继承”），已经开始剖析系统的复杂性了呢，甚至还可以扯上领域驱动设计话语中的“聚合根”——最妙的是，不用思考，信手拈来！

8.2.2.6 自测题



扫 码 或 访 问

http://www.umlchina.com/book/quiz8_2_2.html 完成在线测试,做到全对以获得答案。



1. [单选]

方法学家马宝国老师在推行一种“面向浑元形意太极”的分析方法,按照本书的知识点,看到这个名字,我们最应该想到的是:

- ☐A) 你家老公会劈砖, 小哥会打那太极拳
- ☐B) 马老师的建模方法是伪创新
- ☐C) 马老师的方法假设系统由“浑元形意太极” 构成
- ☐D) 我左手一式太极拳, 右手一剑刺身前

2. [单选]

为什么面向对象分析设计方法比面向过程好?

- ☐A) 面向对象更适合人脑去把握系统的复杂性。
- ☐B) 面向对象和需求的映射更直接。
- ☐C) 面向对象方法更容易掌握。
- ☐D) 面向对象更符合计算机的底层。

3. [单选]

铁路售票处，售票员使用售票系统来售票，在用例进行过程中，系统需要不断向旅客反馈车次、车票和价格信息，系统还需要和银行系统交互。

如果按照本节内容中的三种分析类来识别，"售票"用例的分析序列图的生命线上，会出现_____个边界类，_____个控制类，_____个实体类。

- ☐A) 1, 2, 3
- ☐B) 3, 1, 2
- ☐C) 不定, 1, 3
- ☐D) 3, 1, 不定
- ☐E) 3, 2, 3
- ☐F) 3, 1, 3
- ☐G) 不定, 1, 不定
- ☐H) 3, 3, 3

4. [多选]

关于边界类、控制类和实体类，以下说法正确的有：

- ☐A) 来源于 Grady Booch 对系统的分析类的构思，不属于 UML 规范的一部分。
- ☐B) 实际上就是 MVC 架构模式中的 V、C 和 M。

☐ C) 这三种类可以映射到任何实现架构, 包括 MVP、MVVM、六边形、洋葱型.....

☐ D) 如果所用建模工具没有提供边界类、控制类和实体类的图标, 可以用文本构造型代替或者在类的命名上区分。

5. [单选]

关于系统用例和类, 以下说法正确的是:

☐ A) 系统某个用例的实现中, 使用的类的数量可以为 0。

☐ B) 系统某个用例的实现中, 使用的类的数量至少为 3。

☐ C) 系统某个用例的实现中, 使用的类的数量至少为 1。

☐ D) 系统某个用例的实现中, 尽可能保持一个步骤对应一个类, 必要时可以调整。

8.2.3 提炼类和属性

8.2.3.1 从需求规约之外的其他素材提炼

建模类及类的关系, 可能在软件开发的分析工作流之前就已经发生了。

第 7 章“需求启发”中就提到, 我们在研究资料的时候, 可以通过画类图来整理领域的概念。整理领域概念时, 有时还可以加上状态机图。即使不是为了开发软件, 也可以通过这些手段来整理领域知识, 帮助我们更快地理解和掌握领域知识。

以刘慈欣的《三体 III: 死神永生》中的片段为例, 原文如下, 我

们把其中值得提炼的概念标红：

宇宙的熵在升高，有序度在降低，像平衡鹏那无边无际的黑翅膀，向存在的一切压下来，压下来。可是低熵体不一样，低熵体的熵还在降低，有序度还在上升，像漆黑海面上升起的磷火，这就是意义，最高层的意义，比乐趣的意义层次要高。要维持这种意义，低熵体就必须存在和延续。

至于这意义之塔的更高端，不要去想，想也想不出什么来，还有危险，更不用说意义之塔的塔顶了，可能根本没有塔顶。

回到坐标上来，空间中有许多坐标在穿行，如同母世界的天空中飞翔的矩阵虫。坐标拾取由主核进行，主核吞下空间中弥散的所有信息，中膜的、长膜的和轻膜的，也许有一天还能吞下短膜的。主核记着所有星星的位置，把信息以点阵方式与各种组合的位置模式进行匹配，识别出其中的坐标。据说，主核可以匹配五亿时间颗粒前的位置模式，歌者没有试过，没有意义。在那个遥远的时代，宇宙中的低熵群落比较稀疏，也还都没有进化出隐藏基因和清理基因。而现在——

藏好自己，做好清理。

但所有坐标中，只有一部分是有诚意的。相信没有诚意的坐标常常意味着清理空旷的世界，这样做浪费精力，还有一点点害处，因为这些空世界以后还可能用得着。无诚意坐标的发送者真是不可理喻，它们会得到报应的。

根据以上描述,可以画出如图 8-27 的类图来整理其中的领域知识。整理后,可以看出各个概念之间的关系,还可以看出,领域知识可以分为 4 个部分。

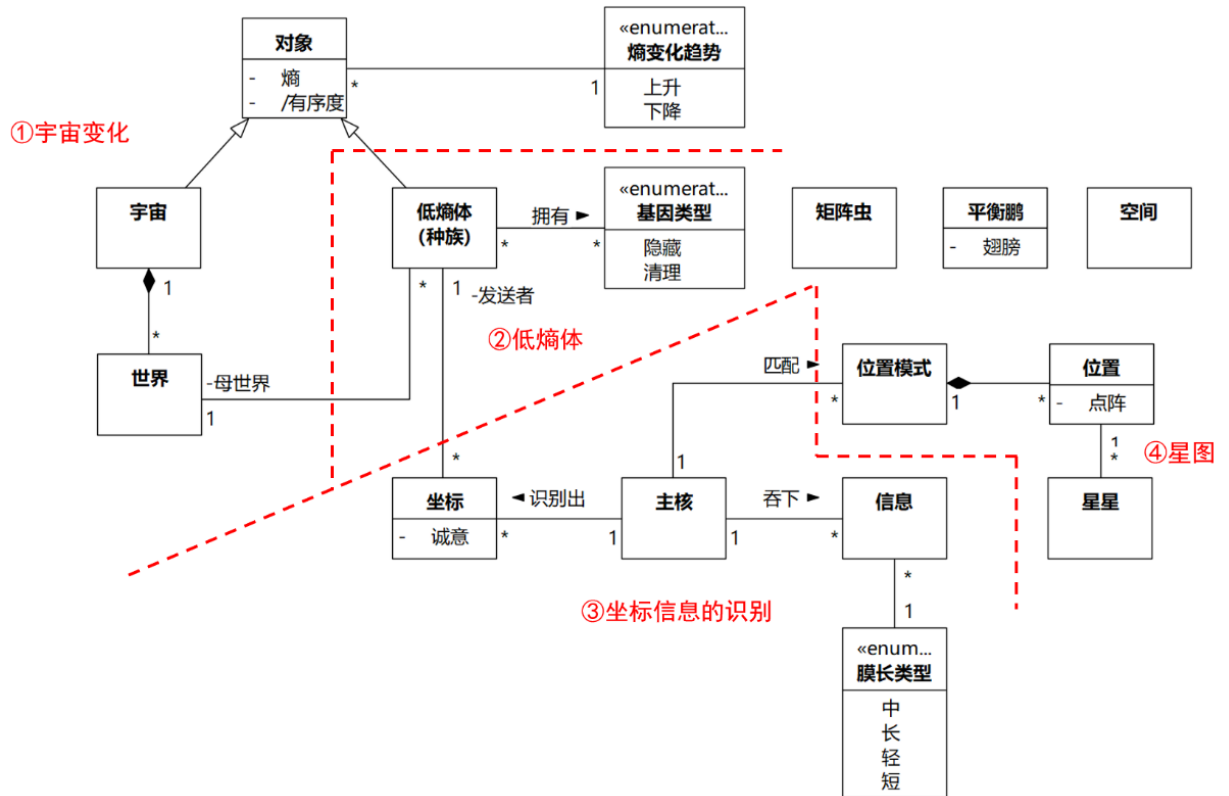


图 8-27 用类图整理《三体 III：死神永生》中的片段

图 8-27 可以称为“宇宙的领域模型”。这个模型可能和信息系统没有关系,因此不能称为“宇宙的核心域模型”或“宇宙的分析模型”。

如果某个信息系统“三体游戏”要封装图 8-27 的领域知识,那么图 8-27 可以称为“三体游戏的核心域模型”或“三体游戏的分析模型”,但不适合称为“三体游戏的领域模型”,因为“三体游戏”中封装了核心域和非核心域的知识。

从上面例子可以看出来,对于以上提到的几个用语,本书是按以下

定义使用的：

领域模型：描述某个领域中的概念及概念之间关系的模型。

分析模型：从核心域视角描述的信息系统的模型。

核心域模型：等同于分析模型。

它们之间的关系如图 8-28。

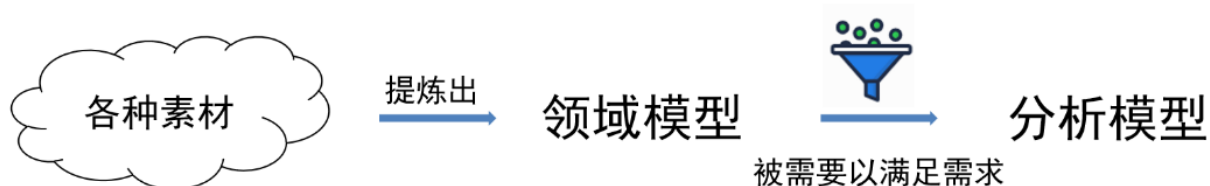


图 8-28 领域模型和分析模型

模型可以用各种表示形式来表示，相应的形式可以叫“领域类图”、“领域 ER 图”、“分析类图”、“分析状态机图”、“分析类文档”、“分析状态机文档”

我们可以从各种素材中提炼某个领域的类和属性，不过这些类只能叫领域类，不一定是合适的分析类。到底哪些领域类会成为目标系统的分析类，需要从目标系统的需求模型（如果用本书上册的方法表示，就是系统用例规约）来判断该系统是否需要这个类。

没有需求规约，目标系统的边界以及应该承担的责任没有理清楚，那么，提炼得到的类到底是不是目标系统将来需要维护的概念，是没法判断的。也就是说，虽然在很多时间点、从很多素材都可以提炼类，让我们在确定分析类时，省去了许多思考，**但分析模型的最终依据只能是需求模型。**

当然，如果你脑子里对于系统该干什么不该干什么清清楚楚，只不过没写出来，那也算是有需求模型了——**不过你得确定你真的懂，而不是拿这个当遮羞布。**

如果按照本书所采用的面向对象建模、UML 表示法和 Ivar Jacoson 三种类的分割，某个领域模型可能会包含某个系统的分析模型中的实体类部分，但不会包含边界类和控制类部分。

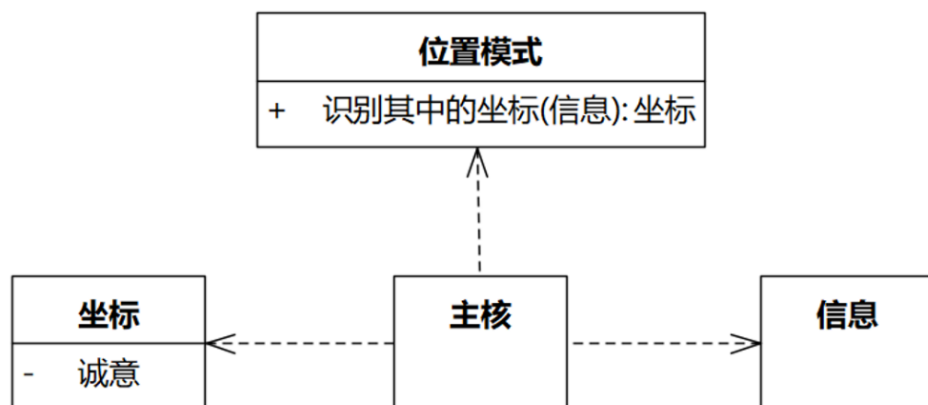
另外，在用类图表达领域模型时，我们只使用泛化、关联等静态关系，不使用动态的依赖关系，也不使用序列图来表达类之间的协作，因为这些内容在相关的信息系统运行时才会出现，而此时我们并没有在构思或建造信息系统。

★注意，刚才说的序列图和第 4 章的业务序列图的区别，也可以参考图 8-25。

例如，图 8-27 中，“主核”和“信息”、“位置模式”以及“坐标”之间是关联关系。这是合理的，因为宇宙系统能把所有关系都记录下来，包括哪个坐标是由哪个主核识别出来的——当然，也包括导致太阳系被降成二维的那几次和三体世界的通信。

但是，如果我们开发一个信息系统“三体游戏”，即使游戏中有“主核识别坐标”的内容，系统未必要记录哪个坐标是由哪个主核识别出来的。此时，“主核”和“信息”、“位置模式”以及“坐标”之间不存在关联关系，只存在动态的依赖关系。某种协作方式（当然可以有其他协作方式）下的类图和序列图可能如图 8-29。

类图



序列图

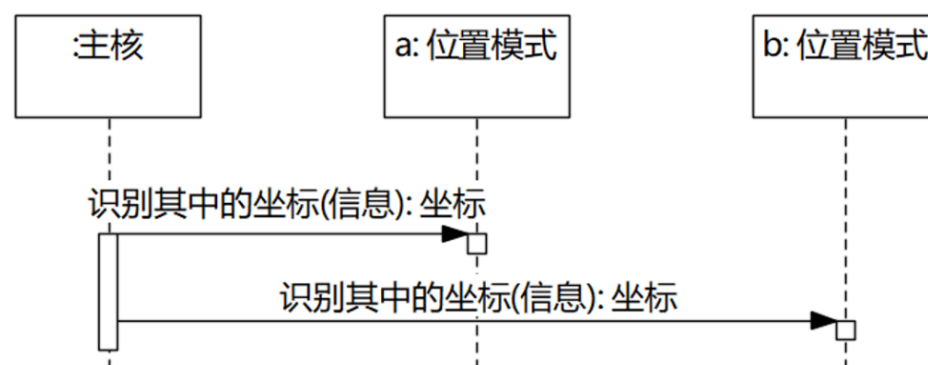


图 8-29 某种协作方式下的类图和序列图

有些书籍和文章作者，对软件开发的工作流没有清晰的概念，把所有用“业务语言”表达的模型，包括组织流程，系统需求规约等，通通叫作“领域模型”。这是不正确的，我们在阅读时要注意分辨。

8.2.3.2 从需求规约提炼

这是提炼类最严格的依据了。

阅读用例规约的基本路径、扩展路径、字段列表和业务规则部分(即所谓“功能需求”部分)，针对表示名词或事件的词汇，逐个思考，**这是不是系统要记住的核心域概念？**如果是，那么它是类，还是某个类的属性？如图 8-30。

用例编号：用例名
执行者

前置条件

后置条件

涉众利益

基本路径

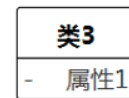
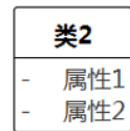
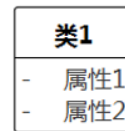
扩展

字段列表

业务规则

非功能需求

设计约束



基本路径：

项目经理【登录】

- 1 项目经理请求预订会议室
- 2 系统反馈项目经理最近所召集会议信息
- 3 项目经理提交要召开的会议信息，请求预订会议室
- 4 系统查找合适的可用会议室
- 5 系统反馈合适的可用会议室信息
- 6 项目经理选择会议室，输入会议主题，确认抢占
- 7 系统将会议室分配给该项目经理组织的本次会议
- 8 系统生成发给会议参加人员的通知，请求办公协作系统发送消息给会议参加人员
- 9 系统反馈会议室已抢占并已通知会议参加人员

扩展路径：

- 4a. 无法找到合适的可用会议室
- 4a1. 系统查找推荐等待的会议室
- 4a2. 系统反馈找不到合适会议室，并推荐等待的会议室

8a. 办公协作系统无响应：

- 8a1. 系统反馈办公协作系统无响应

字段列表：

- 2 反馈信息=会议开始时间+会议参加人员
- 3 提交的会议信息=会议开始时间+会议结束时间+会议参加人员+会议所需设施。
于：视讯设备、电话、投影、白板。

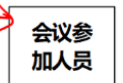
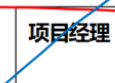
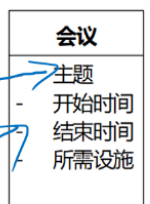
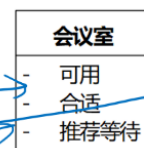


图 8-30 从用例规约识别类和属性

关于“系统要记住的”概念，可以这样思考，系统需要懂得哪些概念以及概念之间的关系，才能根据执行者的请求提供恰当的价值？

如果您有关系数据库建模的经验，也可以这样简单地思考：如果系

统需要维护的信息都采用关系数据库来保存,那么数据库里应该会有哪些表和列? 这样思考得到的表、列和实体类、属性基本上是一一映射的。表对应类,列对应属性,行对应对象,关系对应关联。后面我们会讲到,类模型可以直接转换成关系数据库模型,不需要再花工作量做一遍关系数据库建模。

如果关系数据库建模技能掌握得好,得到的数据模型符合 1NF、2NF 和 3NF,那么用关系数据库建模的思考方式得到的类图极有可能也是合格的。反过来,也有理由怀疑,自诩关系数据库建模能力强的人,如果类建模做得乱七八糟,估计也有吹牛的成分。

当然,我们画类图的目的是整理核心域概念和逻辑,将来映射关系数据库只是顺带的便利。面向对象和关系数据库(或任何的具体存储方式)没有必然的绑定关系。任何系统都可以用面向对象的方式来构造,不管它用什么方式来存储对象。

以电梯为例。我们把为乘客提供电梯服务所需要的所有软硬部件看作一个整体,称为“电梯系统”。

乘客在发出某次召唤之前,电梯系统要懂得各部电梯当前运行方向、当前所处楼层、待去往的目标楼层集合以及楼层的结构,才能够合理应对乘客的召唤。这些可以看作系统要记住的信息。

乘客姓名、乘客召唤电梯的时间、电梯曾经去过的楼层等,系统则不需要知道。

画出类图如图 8-31。

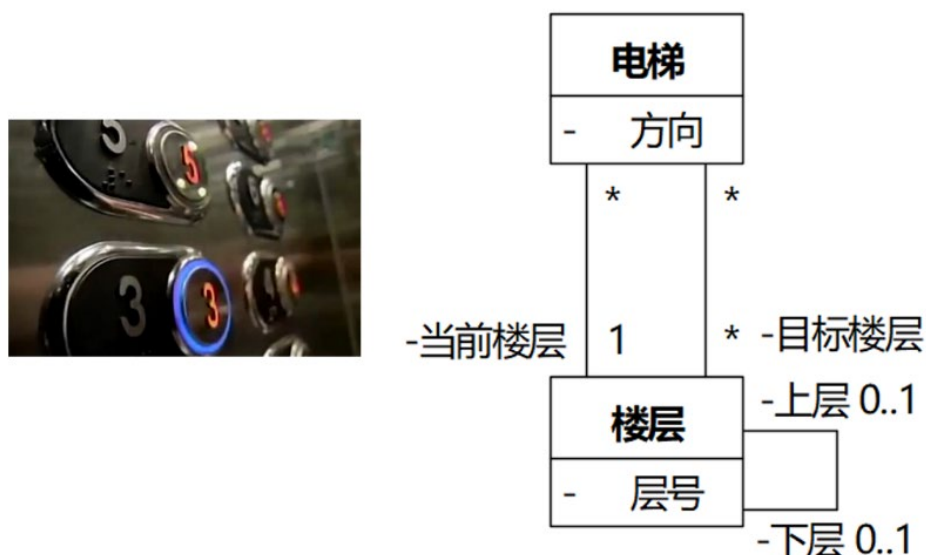


图 8-31 电梯系统可能需要的实体类

这些信息不一定需要被“持久存储”。图 8-31 中的概念中，当前楼层、目标楼层、方向的信息一直在不断变化，甚至在电梯系统停止服务时会被清空，但不影响图 8-31 的类结构。

8.2.3.3 没有需求规约时的思考

有可能你是“敏捷”地做需求，只有一个用例图，甚至连用例图都没有，只是说要做什么功能——当然，做这个功能的理由也是“敏捷”的。

即使是这样的情况，前面所说的思考“系统需要懂得哪些概念以及概念之间的关系，才能根据执行者的请求提供恰当的价值”依然有用，但需要建模人员有更强的抽象能力。

日常工作和生活中，可以有意识训练这方面的能力。针对任何一个系统，我们都可以思考：它需要输入什么，能输出什么，为了能把输入变成输出，系统需要懂得哪些概念以及概念之间的关系？

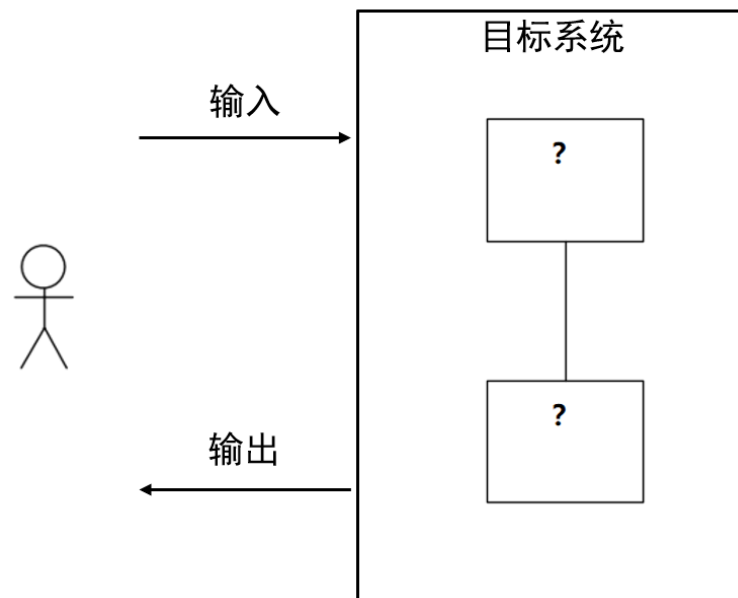


图 8-32 思考系统需要封装的抽象

例如，针对日常生活中的网约车系统，我们使用时，第一个交互可能是这样的：

输入：上车地址和目的地址

输出：各个车辆类型的预估费用。

为了能把输入变成输出，系统需要懂得哪些概念以及概念之间的关系？

要能输出预估费用，需要知道车辆类型的公里价格还有订单要走的路程长度。路程长度需要通过上车地址和目的地址来计算，这个计算又涉及到很多和网约车并非特定相关的复杂概念，由专门的组件来完成是更合适的。思考结果如图 8-33。



图 8-33 网约车系统思考结果

读者可以自行做类似思考，例如，以取款机为研究对象：

- (1) 如果输入为：账号、密码、取款金额，要提供取现金的价值，系统需要懂得哪些概念及关系？
- (2) 如果进一步减少输入，改为输入：账号、密码，系统需要懂得哪些概念及关系？
- (3) 如果把输入改为刷脸，系统需要懂得哪些概念及关系？

如果没有这样的抽象能力，很容易出现“**类图长得像用例图**”的现象。

例如，一个电商系统，开发人员一开始心里有一个想法，要做一个“搜索商品”的功能。那么要实现这个功能，需要什么类呢？开发人员干脆就按看到的表面现象来找类，得到的类图长得非常像用例图，如图 8-34。

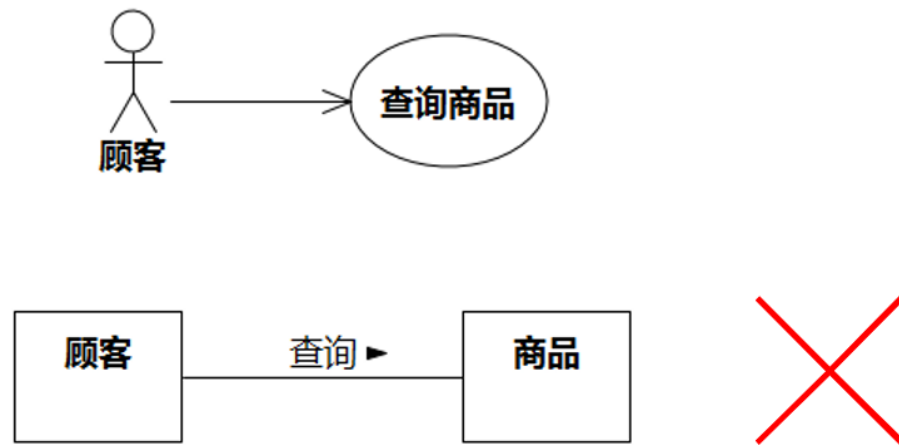


图 8-34 长得像用例图的类图

但是, 系统之所以能够为顾客查询“屏幕尺寸为 6 英寸的 Android 手机”, 不是因为它记住了“哪位顾客查询过什么商品”, 而是因为它记住了各种商品的类别和特征。我们需要的、更合适的类图应该类似于图 8-35。

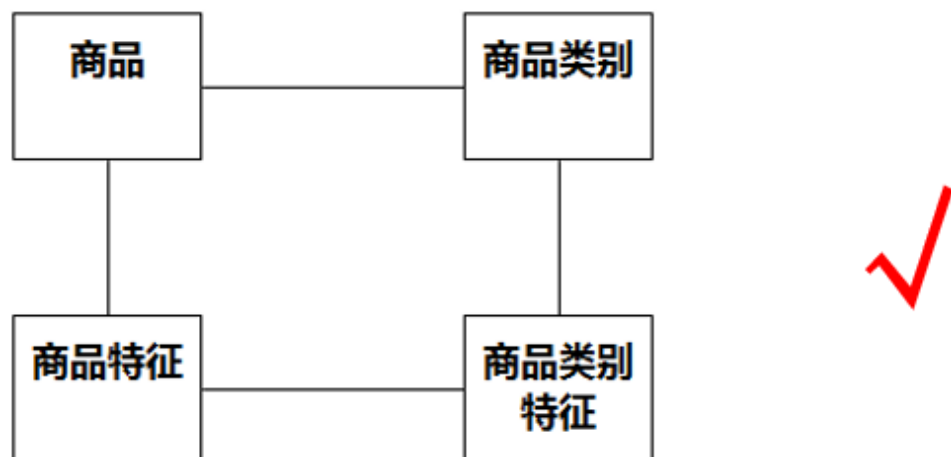


图 8-35 更合适的类图

Martin Fowler 的 "Refactoring: Improving the Design of Existing Code" 第 1 版中, 第 1 章举了一个影片店系统的例子, 从一开始的不合理结构代码, 然后不断重构, 得到更合理的结构。一开始的

类图如图 8-36 所示。

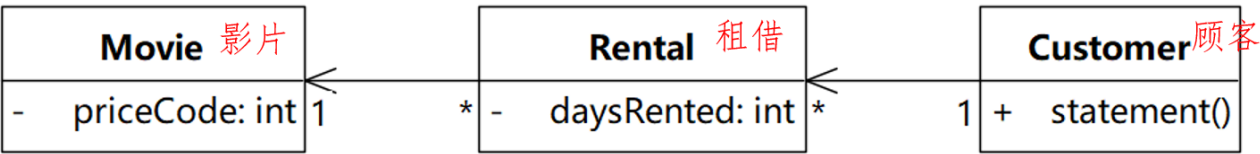


图 8-36 "Refactoring: Improving the Design of Existing Code"中一开始的类图

(没有途径获得清晰的 Refactoring 原书截图, 上图是我按照原书内容绘制, 并加了中文注解。)

图 8-36 的类图长得很像影片店系统的用例图, 画出来像图 8-37:

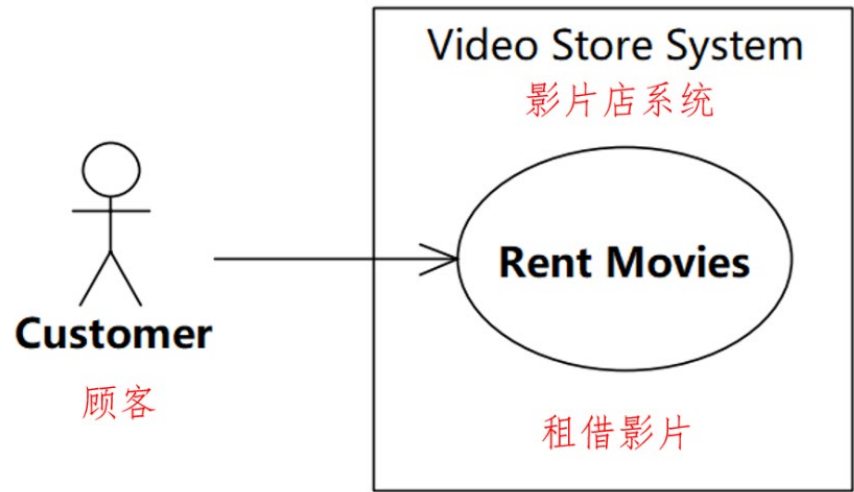


图 8-37 影片店系统的用例图 (根据 Refactoring 书中内容绘制, 原书无此图。)

如果拥有上面所说的思考能力, 看到长得像用例图的类图心里就会响起警铃。影片店系统要提供“租借影片”的价值, 系统需要懂得的关键信息很可能并非哪位顾客租借了哪部影片, 即图 8-36 中的三个类, 而是其他的信息, 如影片的价格体系和库存、顾客的等级。

有了这个警铃,就不用多此一举,先假装弱智写出弱智代码,再“重构”成稍微聪明的样子——这里的强行降智和第 1 章所提到的伪创新“割裂历史”类似。

8.2.3.4 长得像用例图的类图

系统的需求可以看作解决组织问题的解决方案,而系统的类可以看作解决系统需求的解决方案。如果一个解决方案不需要什么思考就可以得到,要么这个解决方案是错的,要么要解决的问题价值已经很小。

长得像用例图的类图,需要的思考非常少,这样的类图很可能带来的价值是非常小的。要么类图错了,要么系统的需求已经没有什么价值。

上文说到,图 8-34 的用例“顾客→查询商品”对应的合适类图是图 8-35。

我们可以进一步思考,如果“哪位顾客在什么时间查询了什么商品”成为系统需要记住的关键信息,那么图 8-38 的类图是合适的:

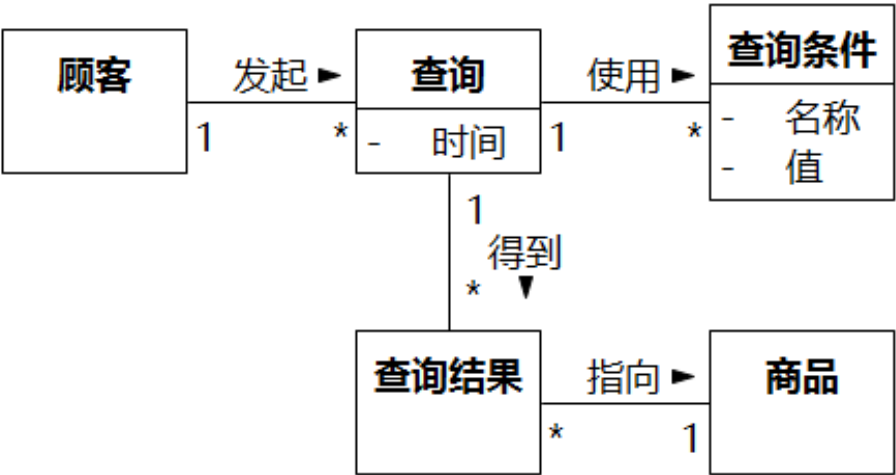


图 8-38 “哪位顾客在什么时间查询了什么商品”成为系统需要记住的关键信息

此时，系统的用例很可能并非图 8-34 中的“顾客→查询商品”，而是类似于图 8-39：

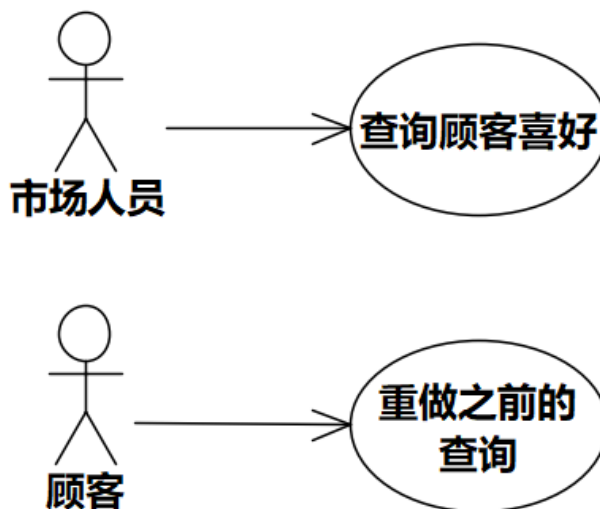


图 8-39 图 8-38 对应的合适用例图

8.2.4 类和属性的命名

8.2.4.1 使用严谨的领域术语命名领域模型中的元素

领域模型中各个元素的名称尽量来自该领域的术语体系。

一个领域之所以能作为“领域”为人认知，必定会在发展过程中沉淀出一套日益完善和精确的术语体系。每个术语有其独特的、其他术语不能替代的含义。

例如物理学中的质量、重量、重力、引力、衰变、裂变、聚变……，各自有各自的含义，不是另一个概念可以取代的。

涉众习惯的用语不一定是严谨的领域术语

也许是出于自己的知识局限，或者出于字数少使用方便，涉众有时

习惯于使用一些不严谨、感性的用语，这些用语不一定适合用来命名领域模型中的元素。

这些用语经常会用颜色、大小等感性认识来称呼领域概念。例如，货车司机可能会把一张单子称为“绿单”，因为单子的颜色是绿色的，但更精确的名称可能是“送货单”；购物时的“小票”，名称来源于面积较小（和面积更大的“发票”比较），更精确的名称可能是“收据”。

这些用语所依赖的信息，稳定性往往比真正的领域内涵要差。有关机构可以改变送货单的颜色，购物收据也可以变成面积比发票要大的大长条，“绿单”、“小票”等用语就不再合适了。即使已经形成了习惯不得不一直沿用下去，真实的情况和字面的意思已经大相径庭。

涉众喜欢用不严谨的用语，这是正常的。某类涉众的领域知识可能会很片面，对领域概念认识不深刻，怎么能寄望一个使用探探来交友的屌丝青年清楚社交六度空间理论呢？

第 7 章说到，涉众关注的是涉众利益，涉众没有资格也没有责任提供需求。同样，涉众也没有（或更没有）资格和责任提供分析。精确使用领域术语是建模人员的责任，不是涉众的责任。

同一领域概念，不同涉众可能会有不同的习惯用语

例如：老鼠-耗子、马铃薯-土豆、祖母-奶奶、宝贝-商品、购物-买东西。

如果怀疑两个用语描述的是同一个概念，可以这样问：有没有不是商品的宝贝？有没有不是宝贝的商品？如果回答都是否，就清除掉其中

一个，否则，应该继续研究两个用语背后的真正含义，必要时在模型中表达其中差别。

如图 8-39, 左侧的几个概念中, 经过审查, 认为"宝贝"和"商品"、"顾客"和"客户"含义相同, 去除其中一个; "用户"、"顾客"、"会员"含义有差别, 在类图上更精细地表达。

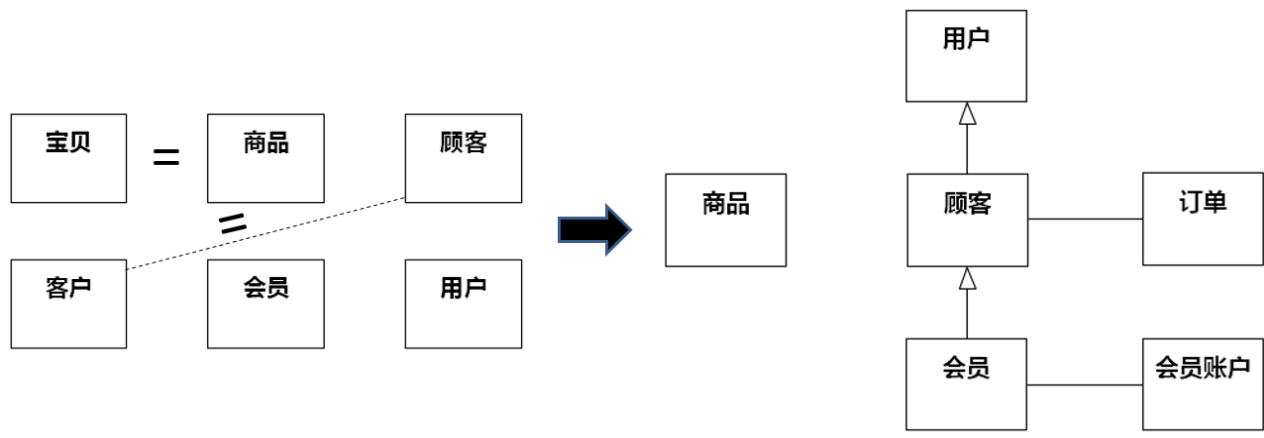


图 8-39 清理冗余的用语

当然，在和人类执行者交互的界面上，针对各自的习惯，依然可以使用不同用语如“宝贝”、“商品”称呼同一概念。

也可以对“同一概念不同涉众用语不同”这部分知识建模，以备在界面上使用，如图 8-40。上面提到的“宝贝”、“商品”就成为了“用语”类的对象。

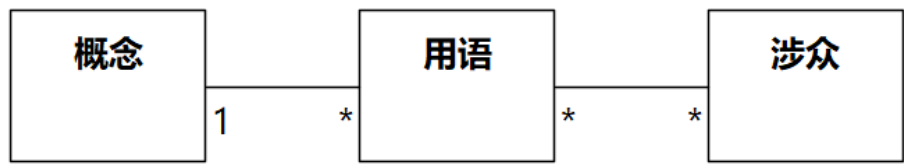


图 8-40 建模“同一概念不同用语”

图 8-40 中所描述的类以及类的关系适用于各个领域，和某个特定

领域并不相关，因此不要把这部分知识和特定领域的知识混在一起。假如建模成图 8-41，就搞混了“是一个”和“有一个”的区别，又变成废话刷工作量了。在后面“识别类的关系”内容中会再详细讲述。

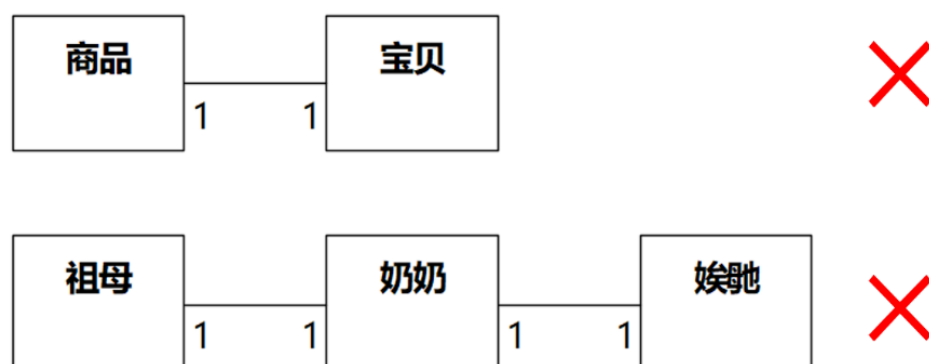


图 8-41 “同一概念不同用语”的错误建模

领域专家

建模人员除了和各类涉众交流，还需要从领域专家那里获取严谨的领域知识和领域术语。

首先要提醒的是，“领域专家”这个词已经被严重污染。

很多人把“经验较多的涉众”当成了领域专家，例如跑运输多年的老司机，例如通过某社交软件约了几百人的老司机。但正如上文所述，这些涉众不一定是领域专家，他们也许能提供丰富的素材，但未必能严谨表达和深入思考。

★得了某个病很多年的老患者，可能并不是这个病的领域专家；没有得这个病但研究这个病很多年的老医生，是这个病的领域专家。

领域专家要能够从研究学问的角度来看待他所在的领域。他应该有该领域比较扎实的基础知识，经常阅读该领域的最新文献，了解该领域

当前研究进展。领域专家可能是该领域的“研究人员”，至少也是该领域的“科普人员”。

“研究人员”、“科普人员”加了引号的意思是他们不一定要有“教授”、“研究员”、“老师”等正式的职位，但至少满足上面所说的要求。当然，上面提到的货运老司机、社交老司机等涉众如果能深入研究自己所在领域的学问，也可以成为领域专家。

领域专家很可能并不是涉众（当然也可以是）。例如，为 A 公司开发供应链系统，建模人员可以向 B 高校研究供应链的学者请教领域知识。学者是领域专家，但他可能跟 A 公司没有关系，这个系统最终会是什么样子，也不涉及到他的特定利益，他并不是涉众。

如果有条件，建模人员可以和领域专家直接交流，借助领域专家的指点，尽快得到确实反映领域内涵的模型。如果没有条件直接交流，可以通过阅读领域专家的各种产出，间接获得帮助。

上文也提到，领域的术语体系是日益完善的。在熟练掌握领域现有术语体系的基础上，辅以严谨的建模技能（背后是一些数学思想），建模人员可以起到“反哺”的作用——帮助清理领域当前术语体系中存在的冗余和矛盾并改善。注意其前提条件：熟练掌握现有术语体系+严谨的建模技能，这不是随便就能达到的！

8.2.4.2 关于 DDD 话语中的“通用语言”

DDD（领域驱动设计）话语中有“通用语言（Ubiquitous Language）”的用语，这是一个伪创新。

(1) 类似“通用语言”的概念早已有之

“术语表 (Glossary)”、“数据字典 (Data Dictionary)” 等内容，几十年前的开发规范中早就已经有了。

下面是出版日期在 Eric Evans 的《领域驱动设计》之前的几本书的截图。

用于描述执行者和用例的术语非常重要。应该在创建组件系统时开发用于描述问题需求和特性的公共和一致的术语表，并且在设计应用系统时使用。对于组件系统的用例模型和用例组

86 第二部分 体系结构风格

件来说，术语表尤其重要，这是因为，通过使用标准和考虑周全的术语，有关人员能够有效沟通并避免误解。

在 XYB 中有很多术语，例如：客户、买方、支付方、现金、电子现金、钱款、工具、支票、贷款、抵押贷款、股票、金融文书、账户、借支、透支、发票、取款、存款、转账、交换、监视等。这些术语之间有着微妙的联系，它们结合使用命名用例、变体和其他组件。如果使用正确的术语命名并描述组件，则复用人员会迅速把需求转换为用例，然后再迅速转换为其他模型中的适当的组件和变体。

精确使用术语有助于复用人员区分和关联变体。使用短语提取钱款、提取现金、提取电子现金使得组件的联系、差别和角色相当明确。

每个组件系统都有自己的术语表。一些术语用来描述从组件系统中输出的组件，会帮助复用人员理解如何使用这些组件。请注意，诸如工具、转账、监视、交换和股票这样的术语在银行领域、网络领域和 XYB 需要处理的其他领域常常有不同的含义。这样，需要在附加在正面的某个文档中描述术语表的子集（例如，如果进行的是传统领域工程，这个术语表应该叫做领域字典）。对于有多个正面的情况，每个正面可以提供一个单独的关键术语文档。与每个正面关联的术语需要仔细选择并标准化，因为复用人员要通过这些术语理解组件可以做什么，并借助它们在各种可替换的变体中作出选择。

这种术语表对于以后开发系统的对象模型也很有用。使用标准术语命名组件和变体，有助于复用人员将组件与用例模型和需求联系起来。对于系统层次的组件系统，术语表中的术语还可以引用复用人员可能需要理解的关键实现特性。

图 8-42 摘自《软件复用：结构、过程和组织》(Ivar Jacobson 等，英文原版出版于 1997 年)

大多数分析者和设计者都很熟悉创建项目词汇表的概念。在项目早期阶段创建词汇表是为了保证每个人都用相同的词汇来表达同一件事情。在 Catalysis 中,类型图是术语表的核心部分——它们代表着专有术语的词汇表,并明确了这些术语之间的关系。该词汇表可以用于项目的所有相关文档,也可以用在程序代码里。

2.1.3 使用静态模型

在开发生命周期的不同阶段,静态模型有不同的作用。如果我们打算从头开始,为旅馆的预订系统提供软件支持,那么分析员提交的第一份文档应该是关于旅馆业务如何运行的描述。类型模型严格定义了词汇表,应该是这篇文档的基础部分。在生命周期的后期,软件中的对象可以用同一种表示法来描述。

当模型被用来分析真实世界时,建模是永无穷尽的。对于一个人,除了他或她住在哪个房间里以外,还有很多可以描述的信息。每个类型图永远可以用更多的细节来扩充,在软件中也是如此。前面我们说过,作为对象的客户,我们感兴趣的是有助于解释所期望的对象行为的模型,而并不关心对象的实际实现。模型可以忽略实现细节,有完全不同于实际实现的结构,只要让客户能够理解实际行为符合什么规律就可以了。

一些工具和作者使用“类图”这个术语。我们只在最细化的设计层次上使用类,表示实际上在代码中是什么。一个类框(标记为«class»)显示了所有直接存储的实例的属性和链接。类型更通用——类型的属性和关联只表达其成员的信息,并没有限定这些信息是如何表达的。相对于类,我们在分析中更多地使用类型,而在设计阶段两者都经常使用。

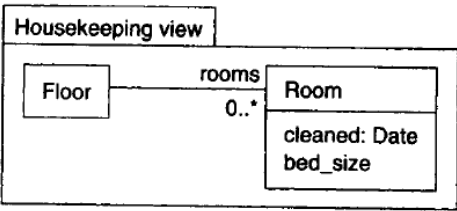


图 8-43 摘自《UML 对象、组件和框架——Catalysis 方法》
(Desmond Francis D' Souza 等, 英文原版出版于 1998 年)

语言的界限就是一个人的世界的界限。

——维特根斯坦

计算机语言会影响你思考问题的方式，以及你看待交流的方式。每种语言都含有一系列特性——比如静态类型与动态类型、早期绑定与迟后绑定、继承模型（单、多或无）这样的时髦话语——所有这些特性都在提示或遮蔽特定的解决方案。头脑里想着 Lisp 设计的解决方案将会产生与基于 C 风格的思考方式而设计的解决方案不同的结果，反之亦然。与此相反——我们认为这更重要——问题领域的语言也可能会提示出编程方案。

我们总是设法使用应用领域的语汇来编写代码（参见 170 页的需求之坑，我们在那里提出要使用项目词汇表）。在某些情况下，我们可以更进一层，采用领域的语汇、语法、语义——语言——实际进行编程。

当你听取某个提议中的系统的用户说明情况时，他们也许能确切地告诉你，系统应怎样工作：

在一组 X.25 线路上侦听由 ABC 规程 12.3 定义的交易，把它们转译成 XYZ 公司的 43B 格式，在卫星上行链路上重新传输，并存储起来，供将来分析使用。

如果用户有一些这样的做了良好限定的陈述，你可以发明一种为应用领域进行了适当剪裁的

程序员修炼之道——从小工到专家

维护词汇表

一旦开始讨论需求，用户和领域专家就会使用对他们有特定含义的术语。例如，他们可能会区分“客户”和“顾客”。于是，再在系统中随意使用这两个词就是不合适的。

要创建并维护项目词汇表（project glossary）——这是定义项目中使用的专用术语和词汇的地方。项目的所有参与者，从最终用户到支持人员，都应该使用这个词汇表，以确保一致性。这就意味着，可以访问词汇表的人员范围应该很广泛——这是采用基于 Web 的文档的一个有效论据（过一会还要进一步讨论这一点）。

提示 54

Use a Project Glossary
使用项目词汇表

程序员修炼之道——从小工到专家

图 8-44 摘自《程序员修炼之道——从小工到专家》（Andrew Hunt 等，英文原版出版于 1999 年）

(2) “通用语言” 迎合了 “广大开发人员” 呆在舒适区的需要

那么，DDD 圈子为什么要重新发明 “通用语言”，并强力吹捧呢？

原因就是第 1 章关于伪创新的内容中阐述的：它迎合了 “广大开发人员” 呆在舒适区的需要。

我们在上文强调，建模人员要虚心学习领域知识，而这是很辛苦的事情，很多人是不愿意吃这个苦的。

“通用语言” 妙就妙在它告诉你，可以不用吃苦不用走出舒适区！

我们先来看一下 Eric Evans 在《领域驱动设计》中对 “通用语言” 的陈述，如图 8-45。

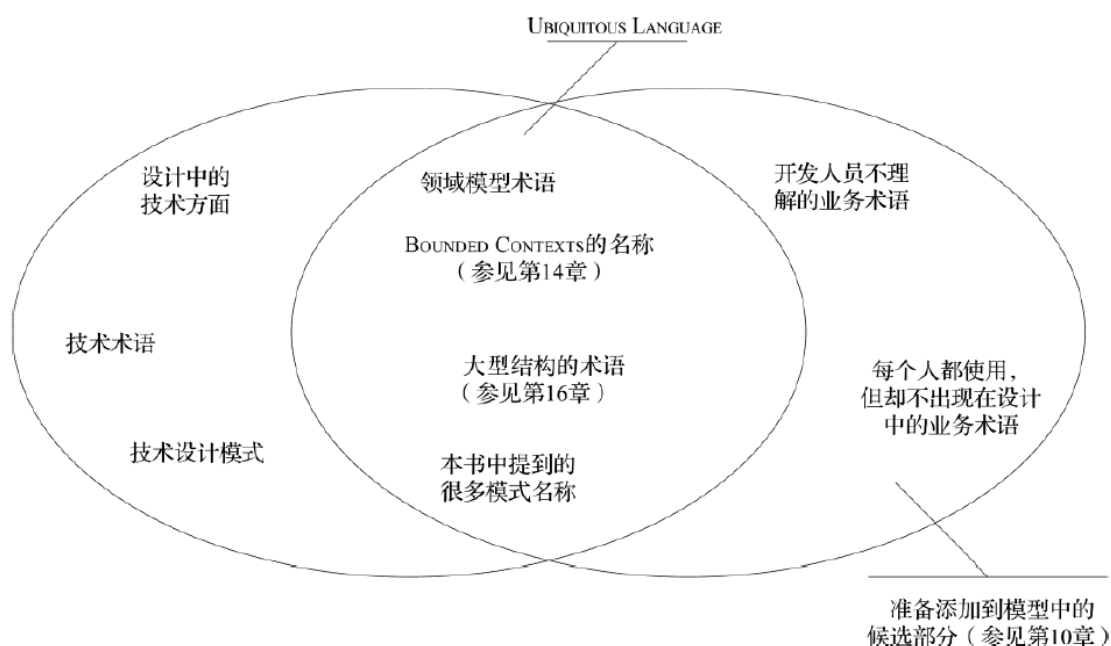


图2-3 术语的交集产生了UBIQUITOUS LANGUAGE

图 8-45 摘自《领域驱动设计》(Eric Evans, 英文原版出版于 2003 年)

从图 8-45 我们可以看到，“通用语言” 是 “技术术语” 和 “业务

术语”的交集，用本书的话来说，就是“非核心域术语”和“核心域术语”的交集。

问题是，有交集吗？

技术术语（非核心域术语），例如浏览器相关的术语：请求、回应、渲染、DOM、序列化、反序列化.....

业务术语（核心域术语），例如商场相关的术语：商品、顾客、订单、库存、收银、盘点.....

“技术术语”和“业务术语”哪里有交集？莫非是把这些正交的术语随意组合，以达到废话刷工作量的效果？此处可以回顾我们在 **8.1.3 域之间映射和协作的套路**中关于 $a \times b \times c$ 的陈述。

DDD 话语中所谓的“技术术语”，根本和“技术”无关，其实只是“技术人员”在没有虚心学习领域知识的情况下编造的“业务术语”——“技术人员编造的业务术语”。

而“通用语言”使得“技术人员”编造“业务术语”变得理直气壮。这是一个大倒退。

现实中，难免会有“技术人员”懒得去调研涉众，懒得去学习领域知识，乱做一通了事的现象。这是人性使然，很难杜绝，但至少我们还知道这样做是不好的，如果要做更好，应该怎么做。

如果把偷懒变成理直气壮，味道就变了。

就像我们说“人是自私的”，这是低调描述一个事实，但如果理直气壮地说“人不为己，天诛地灭”，味道就不一样了。

图 8-46 是 Vaughn Vernon 在《实现领域驱动设计》中的陈述：

那么，你认为你已经知道了什么是通用语言了？

很明显，通用语言是一种业务语言。

抱歉，不是。

通用语言必须采用工业标准术语。

不完全是。。

通用语言是领域专家专用的。

对不起，不是。

通用语言是团队自己创建的公用语言，团队中同时包含领域专家和软件开发人员。

对了。

图 8-46 摘自《实现领域驱动设计》(Vaughn Vernon, 英文原版
出版于 2013 年)

看，不是业务语言，不是工业标准（Industry Standard，译为行业标准更好）术语，而是团队自己创建的！也就是说，同一个领域，开发团队不同，团队里的人不同，所得到的“通用语言”可能就不一样。

观测会影响结果，这莫非是软件开发的“薛定谔的猫”？你有你的物理学，我有我的物理学，本开发团队自有“队情”在，“技术人员”这小日子过得可真是舒坦啊！

更妙的是，“技术人员”还会自我感动。本来我是高大上的“技术”，现在向你“业务”开了个口子，让你也参与进来，你“业务”应该感恩戴德了！

这是一种智力上的优越感所带来的傲慢（当然还有金钱、Quan 力，不便展开，就不提了）。

如果某个领域的从业人员的平均智力水平（学历、学校、智商等）不如软件开发行业，那么软件开发人员，即所谓的“技术人员”在面对该领域的“业务人员”时，是有一种优越感的。

这种优越感让“技术人员”在做供应链系统、商场系统时有足够的底气来编造“通用语言”，因为他觉得货车司机、仓管、外卖小哥、商场经理的智力水平不如他。

如果所开发系统的核心域是凝聚态物理、非线性分析之类的，“技术人员”面对智力水平超过他的“业务人员”，这份优越感就不会有了。这时，“技术人员”可能就会虚心去学习相关知识，因为他觉得这是一件高大上的事情，有面子！

(3) 不要让利益博弈压倒“客观规律”

有一种论调值得警惕。该论调认为“通用语言”让“技术人员”一方有了话语权，不用受“业务人员”一方主导，不用低三下四地去学习领域知识，这对“技术人员”一方有好处。

这样的论调是利益博弈压倒了领域的“客观规律”，不可取。

如果系统的核心域模型没有准确体现领域的“客观规律”，而是发生较大的偏离，那么最终的结果必然是该系统容易出错或者应变成本很高。这种情况也许对某些摸鱼的开发人员有“多劳多得”的好处，但对于整个开发团队以及涉众来说，肯定是有害的。

注意，上面说的“客观规律”加了引号，意思只是说，这些规律不会受开发团队、实现技术等变化的影响，不代表这些规律符合科学。

最常见的，游戏中的知识体系和各种规律，像王者荣耀中的攻击、防御、移动，是魔法不是科学，但建模人员依然要认真去学习和体会这一套体系中的各种“学问”。

同理，上文提到，建模技能可以帮助清理术语中的冗余和矛盾，但仅止于此，建模技能并不能帮助判断该领域的知识是否科学。

(4) “语言” 过于宏大

“通用语言”的“语言 (Language)”这个词太大。语言要有自己的语法，汉语算，C 算，UML 也算，“通用语言”哪里有？术语集或术语表的称呼更合适。

8.2.4.3 核心域透镜

在为了软件开发而建模时，建模人员可能会用自己熟悉的非核心域术语来代替不那么熟悉的核心域术语，还引以为豪。例如，面对一段集装箱领域装箱规则的描述，建模人员立即在大脑中把它转换成自己熟悉的概念：栈、链表、树.....而且认为这是“透过现象看本质”，甚至宣称“我就是程序，程序就是我”！

Fred Brooks 在《人月神话》中引用了 James Coggins 的一段话：

The problem is that programmers in 0-0 have been experimenting in incestuous applications and aiming low in abstraction, instead of high. For example, they have been building classes such as linked-list or set instead of classes such as user-interface or radiation beam or finite-element model.

问题是面向对象程序员在开发错综复杂的应用时,关注的是低层次,而不是高层次的抽象。例如,他们开发了很多像链表或集合这样的类,而不是用户界面、射线束或者有限元模型。

不同领域有不同的难题。不能因为觉得困难就想方设法逃避,对真正要解决的核心域问题视而不见,却花精力去做那些自己熟悉的、他人已解决的非核心域问题。

为了避免核心域概念被非核心域概念掩盖,我们可以采用一种如图 8-47 所示的“核心域透镜”的思考方式:如果从核心域的视角去看这个概念,或者说把这个概念映射到核心域,我们应该得到什么概念?

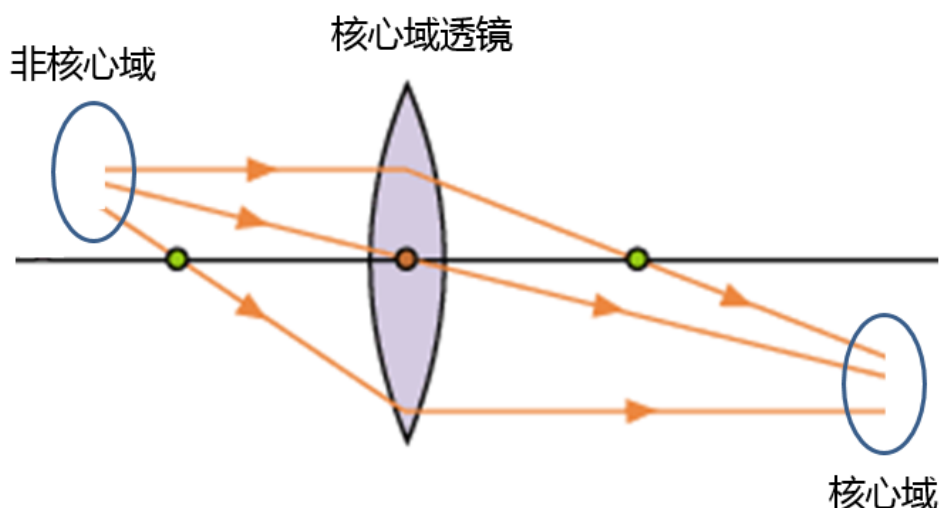


图 8-47 用核心域透镜映射各种概念

例如,以“学习和考试”作为核心域,经过透镜前后的概念对比如图 8-48。

非核心域			核心域透镜	核心域
原描述	原描述过去变体	原描述将来变体 (猜	学	映射后的核心域概

		想)	习	念
PowerPoint	黑板、玻璃幻灯片、 赛璐珞幻灯片	全息	和 考 试	演示工具
检查 IP 地址	看脸、看签名	检查另一种协议地 址 检查大脑芯片标识		检查重复听课学生
点击“开始”按钮	观察到考生开始书 写	?		开始考试
向数据库“答题”表添加一条 答题记录	在答卷上涂黑一格	?		答题

图 8-48 经过透镜前后的概念对比

对每一个用语我们都可以这样过一下:这个用语属于核心域概念吗?如果不属于,映射到核心域概念意味着什么?

例如,图 8-49 左侧,“商品”有一个“加粗显示”的属性来标记它是否加粗显示。如果核心域是“商品”相关的领域,那么“加粗显示”不属于核心域的概念。我们可以追问:为什么要“加粗显示”,能在核心域找到原因吗?回答可能是:因为该“商品”是“热销商品”。“热销”才是核心域概念,可以改成图 8-49 右侧。

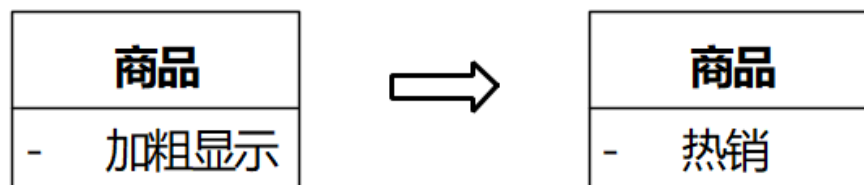


图 8-49 “加粗显示”的映射

这里涉及到两个知识：

知识一：某商品是热销商品。这是商品领域的知识，已经存在很多年。

知识二：如何表示“热销”的概念。这和表示方式、时代审美、展示的目标人群等有关。图形接口的表示和文本接口、语音接口不同，十年前的图形接口和现在的图形接口不同，面向年轻人的图形接口和面向老年人的图形接口不同……

8.2.4.4 命名中不带冗余内容

如果把模型元素命名中的某个部分删去，不影响建模人员对该模型元素的认识，那么这个部分没有必要存在。

如图 8-50 中，在“人员”后加一个“类”字，实际上也是把两个不同领域的知识叠加在一起。

知识一：人员是一个类。

知识二：用 UML 表示法的图形表示，类是一个方框。

知识一和知识二是正交的。在对知识二已经有了共识的情况下，再加上一个“类”字是冗余的。

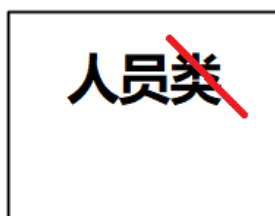


图 8-50 删除命名中的冗余部分

关于类和属性的命名，常犯的冗余错误有：

- (1) 在类名的最后加"类"字或在类名的前面加"Class"或"C"；
- (2) 在类名中加入“领域”、“实体”等词；
- (3) 在类名的最后加"情况"、"信息"、"记录"、"数据"、"表"、"库"、"单"等；
- (4) 在属性名前加类名。

如图 8-51，按"类的属性"念出来，"人员的姓名"很好，"人员的人员姓名"就冗余了。



图 8-51 属性名称前不需要加类名

(5) 给类加上标识属性

对象有标识，这是一个共识，不需要为类专门加一个如 "**ID" 之类的标识属性。

如何表达对象的标识,这是另一个领域的知识,其实现规律和当前所研究领域的知识无关(除非当前所研究领域就是“如何实现对象标识”的领域)。一旦确定实现的套路,在设计工作流通过人工或工具按照套路加上即可。

此处说的标识是**为了区分对象而添加的标识**,在设计工作流中可以以整数、GUID 等各种方式实现。它应该不带任何领域知识,因为一旦带有领域知识,就意味着一旦领域知识发生变化,它也要跟着变化。

除了标识之外,可能还有其他在类的对象集合内值唯一的“编号”属性,如订单编号、人员身份证号、房间号等。这些“编号”属性往往带有领域知识,例如房间号“203”会暗示这个房间是2楼第3个房间。

这样的暗示,是为了**让人方便**记忆和识别,计算机并不需要标识中含有领域知识,除非对象没有其他属性,而是把各种领域知识都凝结在标识中,需要计算机来解析。

在设计工作流,虽然这样的属性也可以作为标识使用,但尽量不要这样做,因为其中的领域知识一旦发生变化,就会带来风险。

本书作者的身份证号,最开始是15位的“3401**74*****1”,1999年升位,变成了18位的“3401**1974*****19”。身份证号中暗示了许多知识:3401**→安徽省合肥市蜀山区,1974****→1974年**月**日生,**1→当日出生的男性顺序号,9→校验码。这些信息发生变化时,原有的暗示就失效了。

应该把这样的“编号”属性当作普通的属性,把它和对象标识区分开。“编号”属性带有领域含义,而且不是每个类都需要,因此需要出

现在分析类中，而标识不带领域含义，是所有类都有的，因此不需要出现在分析类中。

对象标识仅用于标识对象以及在非人系统或系统内部组件之间传递，不需要在人机交互的界面上出现。也就是说，不需要人类执行者输入对象标识，也不需要向人类执行者展示对象标识。如果需要，在人机交互的界面上出现的应该是“编号”属性。

(6) 给类加上名为“状态”的属性

很多建模人员会自作聪明地给类加一个名叫“状态”的属性，如图 8-52，还洋洋得意，“哇塞，我知道这个类有状态噢，我真厉害!”。

和对象有标识一样，对象有状态，这是面向对象这个领域的共识，和“人员”等具体的类没有特定关系。如果加上去，又变成了批量刷废话。

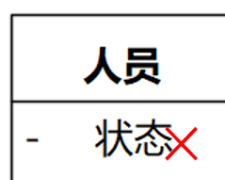


图 8-52 不需要名为“状态”的属性

至于类的状态如何实现，是像图 8-52 那样添加一个名为“状态”的属性，还是采用其他的实现方式，属于另外一个领域的问题。

注意区分名为“状态”的属性和状态属性。如图 8-53，“人员”有一个“已婚”的状态属性（类型可能是 bool），这背后包含特定的领域知识，可以暂时留在那里。当然，后面会引入状态机，将类的状态属性变成状态机中的状态，然后从类的属性列表中清除它们。



图 8-53 状态属性

关于状态机的建模以及实现，后面章节再详细描述。

如果犯了以上所列举的错误,相当于强行把正交的知识叠加在一起,往往就会得到“废话刷工作量”的结果——记住这个要诀,如果有一天读者想要故意“废话刷工作量”,就把它用上。

举个“废话刷工作量”的例子以加深印象,如图 8-54,如何“废话刷工作量”?

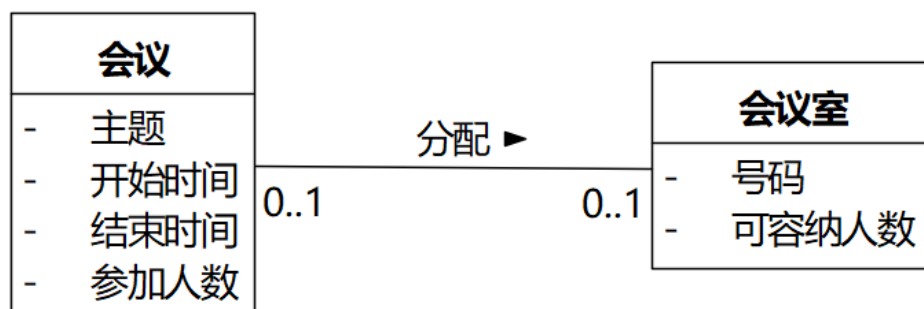


图 8-54 无冗余的类图，等待“废话刷工作量”

按照上面提到的冗余错误，一个个用上去，可以得到“废话刷工作量”的步骤如下：

(1) 因为方框上部是类名，下部是属性，连线是关联，所以，给类和关联的名称加上“类”、“属性”、“关联”。

(2) 可以看出，这些类是领域类或分析类，所以，把“领域”加

到类名称后面。

(3) 因为这些类都代表了一些信息，所以，把“信息”加到类和属性名中。

(4) 因为类的属性是“类的”属性，所以，给属性前面加上类的名称；因为关联也是属性的一种，只不过类型是另一个类，所以，每个类为关联加一个属性。

(5) 因为每个对象都有标识，所以，给每个类加上 ID 属性。

(6) 因为每个对象都有状态，所以，给每个类加上名为“状态”的属性。

(关联、状态等相关知识，后文讲述。)

演变过程如图 8-55。

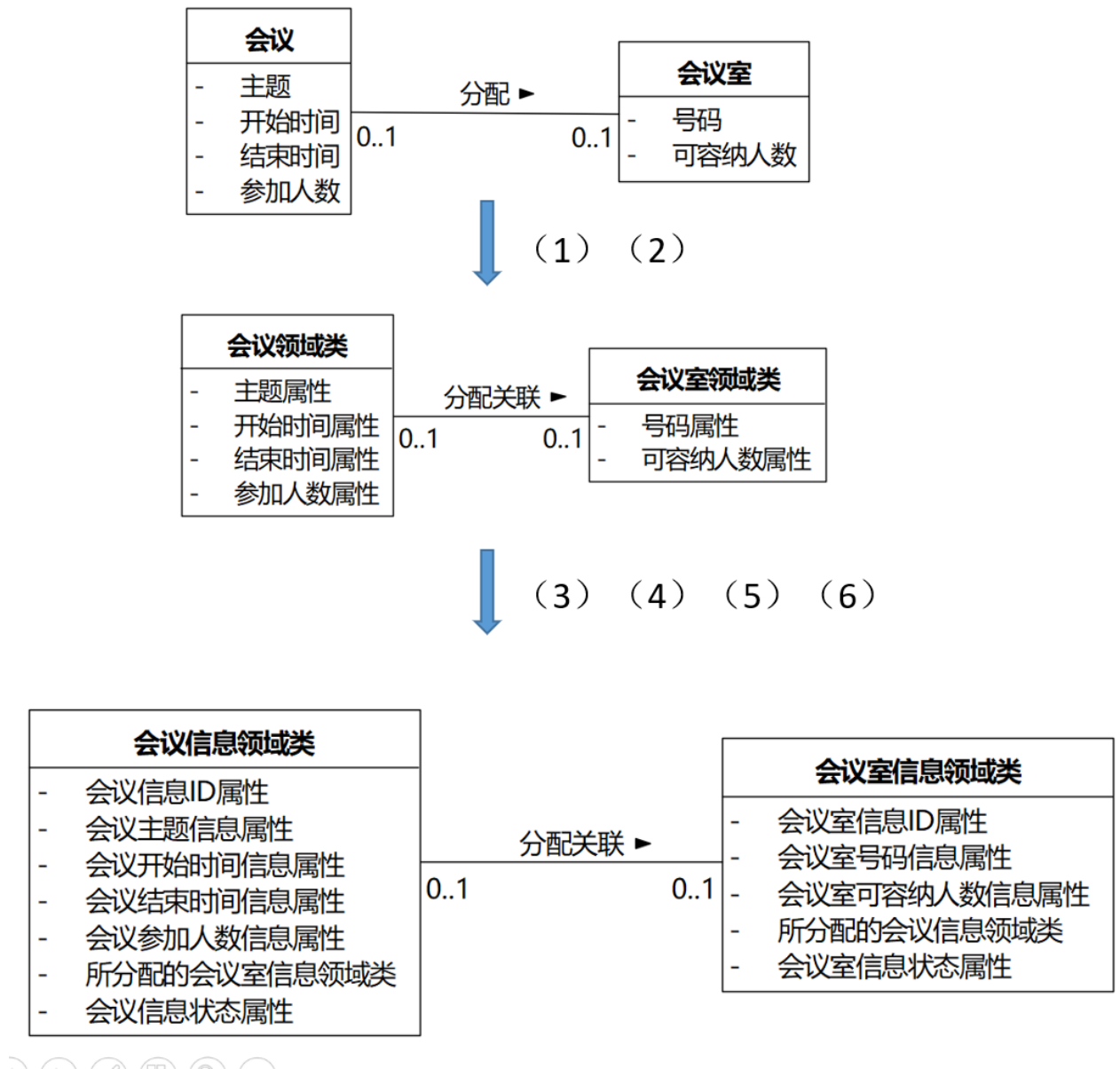


图 8-55 废话刷工作量的过程

类图上的字从最开始图 8-55 顶部图形的 28 字增加到底部图形的 135 字，但没有增加任何有价值的信息。

“信息”、“数据”等也不是不可以作为类的名称。如果系统关注的焦点是“信息处理”，处理的信息是什么内容无所谓，“信息”、“数据”也可以作为类的名称，类图可能如图 8-56。

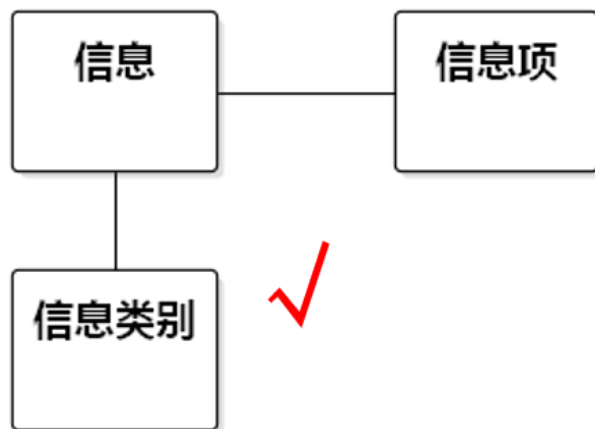


图 8-56 "信息"作为一个类的名称

类的名字就叫“信息”，同样也不能像图 8-55 那样出现一个“会议信息”。“会议”和“信息”不会出现在同一个抽象级别。

8.2.4.5 命名的词性

类和属性应该用名词命名。上文出现的类“人员”、“商品”、“姓名”等就是名词。

在汉语中，动词可以不做任何变化，直接作为名词使用。例如，“我的奋斗”、“嫌疑人 X 的献身”以及“面向对象设计”，其中的“奋斗”、“献身”和“设计”就是动词的名词化。

因此，“申请”、“审批”、“入库”、“出库”等同样可以作为类的名称。虽然它们看起来是动词，但在作为类的名称的时候，已经被当作名词了。

有的建模人员不理解这一点，总觉得看着不舒服，非得在后面加一个“单”、“记录”、“信息”、“事件”之类，变成“申请单”、“申请记录”、“申请信息”、“申请事件”。

其实并不需要，否则每个动词刷一个后缀，又变成废话刷工作量了。

“事件”还说得过去，像“单”、“记录”、“信息”等后缀已经假设了其具体形态，就更不可取了。即使建模时面对的素材是一张纸质的申请单，类的名字也应该写“申请”，因为这才是本质的概念。

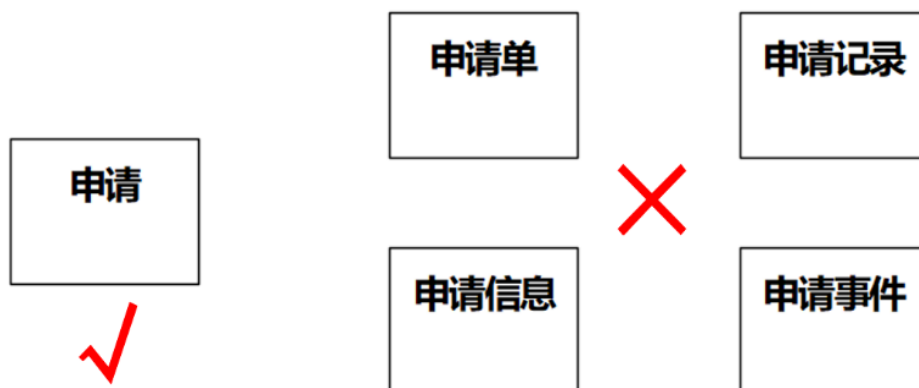


图 8-57 “动词”直接作为类的名字，不用加后缀

如果一个用语在某个领域中已经存在很久，成为了该领域的术语，即使它看起来犯了以上提到的冗余错误，用来作为模型元素的命名也无妨。例如“订单”带有“单”字，实际上描述的是一次“购买”或“交易”，不过“订单”已经在领域中广泛使用，而且把“单”去掉留下“订”也不合适，所以“订单”可以作为类名。

在其他语言中，动词不一定能直接用作名词，可能要做一些变化。

在英语中，像“设计”是可以直接使用的：design→design，但像“献身”可能就要加一个词缀：devote→devotion。其他词缀可能还有 ment、ance、ing、ness 等。

★领域驱动设计话语中的“领域事件 (Domain Event)”采用了动词的过去式来命名，例如 *OrderShipped*，这是错误的。应该用名词命名，关于这方面的阐述，可参见我写的这篇文章《DDD 话语批评之

一：评“状态和事件本质相同”》，
<https://mp.weixin.qq.com/s/QhAhSdET5psZQEEW6f9KoA>

因此，在汉语中，“支付”可以作为类的名称（名词），也可以作为操作的名称（动词），而在英语中，类的名称可能叫 Payment，操作的名称叫 Pay，如图 8-58。

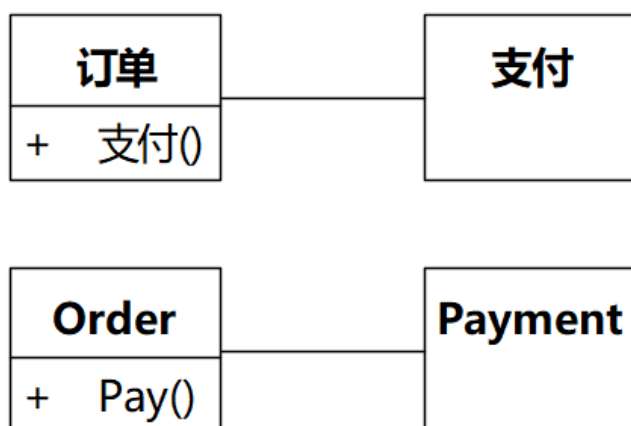


图 8-58 汉语和英语的类名和操作名

还可以看出，由于英语名词后面加了词缀，像在汉语中忍不住加一个“记录”变成“支付记录”那样，在后面再加一个“Records”之类的概率小了很多。

但也不是所有时候都忍得住，例如，图 8-59 是“Domain-Driven Design: Tackling Complexity in the Heart of Software”中的一张类图，其中，Delivery 和 Handling 已经是名词，History 和 Event 没有带来有价值的额外信息，可以删掉。

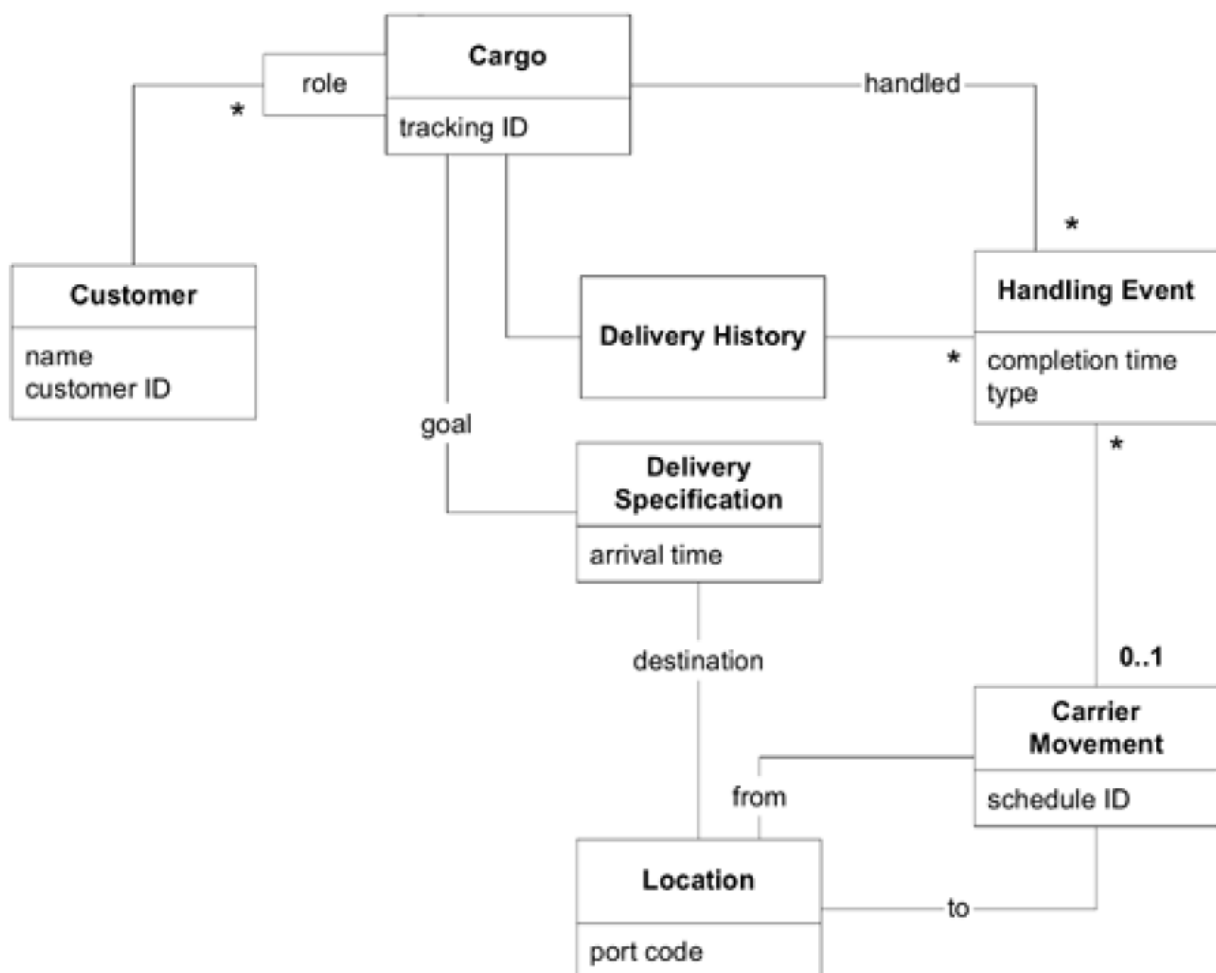


图 8-59 摘自 “Domain-Driven Design: Tackling Complexity in the Heart of Software”

形容词也可以名词化，例如，“沉默是金”、“把悲伤留给自己”。

但是，和动词不同的是，即使把形容词当名词使用。这个词也不会直接用作类和属性的名称。也就是说，“沉默”、“悲伤”不会成为类和属性的名称。

对于很多形容词，领域中另有术语用于测量它。例如，测量“重”，有“重量”，测量“浓”，有“浓度”。“重量”、“浓度”等词就可以作为类和属性的名称。

如果某些形容词，像上面的“悲伤”，领域知识中还没有术语用于测量它，需不需要软件开发人员造一个？例如，在形容词后面加个“值”、“度”、“指数”等，“悲伤”→“悲伤度”？

不需要，也没资格。

如果领域知识中没有测量“悲伤”的术语，说明这方面的领域知识可能还不存在，怎么可能会让软件开发人员在软件中封装这些知识呢？

如果涉众硬是需要软件开发人员在软件中封装这些知识，那么此时软件开发人员就不只是软件开发人员了，他要扮演两个角色：首先，他是一名研究“悲伤”的学问的领域专家；其次，他是一名把某个领域专家的研究结果封装到软件中的软件开发人员。这两个角色可以由同一个人担任，也可以分开。

★同前文所述，此处的研究不一定是科学研究，也可以是修仙、漫威、哈利波特、王者荣耀。

形容词可以作为类的状态的名称。当还没有给类添加状态机时，形容词可以作为状态属性暂时在类中出现，但要注意，状态属性并不是真正的属性，有了状态机之后，要将其删除。

如图 8-60，“重量”是“人”的属性，出现在上方的类框中，“重”是“人”的状态，出现在下方的“人”状态机中。从状态机图上看，状态“轻”、“重”代表了“重量”属性的一个取值区间。

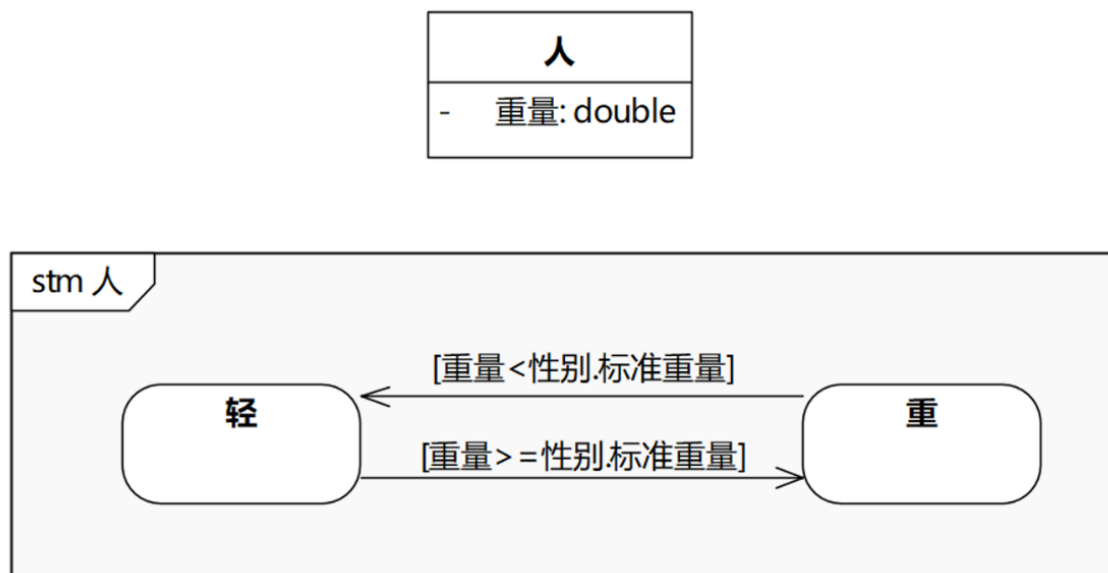


图 8-60 状态属性和测量值

在实现状态机时，有很多实现方式可以选择，其中有一种实现选择是，把“重”作为一个布尔类型的状态属性，重新放回“人”这个类中。

关于状态和状态机建模，后文会再详述。

8.2.4.6 命名所用的语言

这里说的不是编程语言，而是汉语、英语、日语……

给核心域元素命名，使用的语言应该首先考虑精确体现核心域内涵和方便开发团队思考和交流核心域知识。该用汉语就用汉语，该用英语就用英语，该用日语就用日语。

以前经常会考虑转换到编程语言时需要改名的问题。在设计工作流，如果我们使用的编程语言只能用英语命名类、属性、操作等——更严谨的说法应该是编译器广泛支持的字符集比较小，那么还需要一个对编程语言合法的名字。

建模工具例如 EA，一般会提供别名（Alias），真实名称用编译器

支持的字符集，再加一个别名用于显示。

随着时代的发展，编译器、DBMS 等支持的字符集越来越大，上面提到的问题慢慢不再是问题。如果团队成员更熟悉汉语，那么在代码中使用汉语命名即可，省去蹩脚英语或者模糊的汉语拼音缩写（啥？你说“可以加注释嘛”？）给后续工作带来的麻烦。

其他如在文本编辑器中编码时切换输入法、版式不好看、中文标点符号等问题，这和特定编程语言的语法有关，如果能尽量在图形建模时表达核心域逻辑，生成尽可能多的编程语言文本，这方面的麻烦就会小很多。

以上所说仅是针对目标系统的核心域元素的命名，不涉及非核心域部分。非核心域部分内容，和计算机和软件领域有关。这方面的“外语”，不管是计算机和软件专业的各个概念，还是编程语言的关键字，要求软件人员熟练掌握是理所应当的。

例如，浑元形意太极掌门马老师找我们为他的门派搞一套“修炼辅助系统”。作为软件开发人员，能认真去体会马老师说的“化劲”、“武德”、“真气”、“掌门”之类的概念就已经不易，还要把它们变成英语就更难为人了，估计马老师自己也不知道。

但是，作为软件开发人员，不知道什么叫 DTO，什么叫 Iterator 就不应该了，因此，命名为“武德 DTO”之类是没有问题的。

如果目标系统是面向国际的，那怎么办呢？还是前面说的，首先考虑精确体现核心域内涵和方便开发团队思考和交流核心域知识。该用汉语就用汉语，该用英语就用英语，该用日语就用日语。

8.2.4.7 类命名用单数

类的名称已经是一个抽象概念,既代表属于这个类的所有对象的集合,也指代集合中的任何一个对象。

例如,“人”这个概念既代表所有符合“人”的特征的对象的集合,我们可以说“人是一种动物”;同时,“人”可以代表“任何一个人的个体”,我们可以说“人有姓名、身高”。这两种用法,分别对应于后文要讲解的泛化和关联关系。

如果用复数表达,例如汉语“人们”,英语“people”,第二种用法就很别扭了,实例“某个人们”是什么?特别是在和其他类关联时,如果本方多重性为多,又该如何处理?

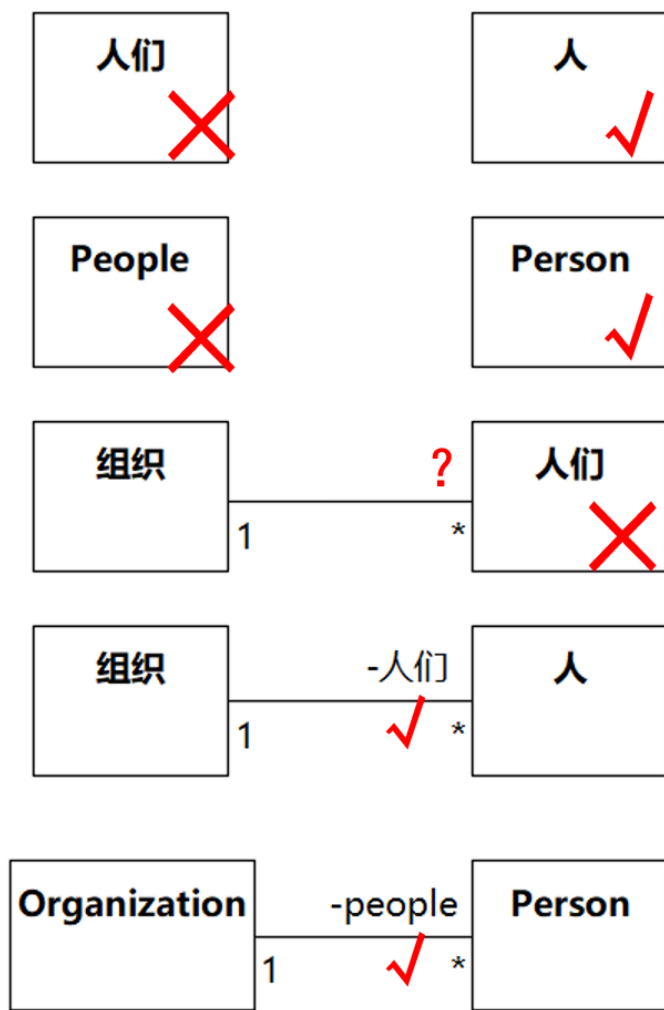


图 8-61 类命名用单数

如果“某个人们”另有含义，那么应该有另外一个类。例如，图 8-61 中的“组织”。“组织”和“人”关联，“人”一端的多重性为多，此时，“人”一端的角色名，也就是“组织”中类型为“人”的属性名，可以用复数“人们”或“people”代表“人”的集合。

有一些常见的开发习惯，如数据库表名用复数，甚至有的框架在类转换表时，直接就在名称后面加上 s，理由是表里有很多行。其实“类”、“表”的概念已经隐含了“多个对象”、“多行”的意思，不用再加了。而且，

如果英语不熟，还得费心思去想正确的复数形式，何必呢？

8.2.4.8 自测题



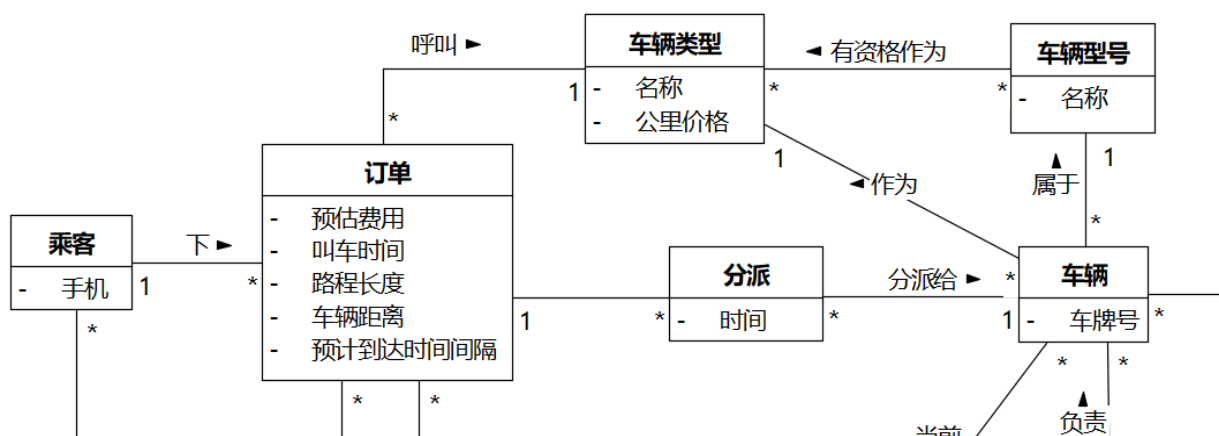
扫二维码或访问

http://www.umlchina.com/book/quiz8_2_4.html 完成在线测试，做到全对以获得答案。



1. [单选]

针对下图，以下描述最正确的是：



- ☐ A) 这是“打车”领域的分析类图。
- ☐ B) 这是“滴滴打车”系统的核心域类图。
- ☐ C) 这是“打车”领域的核心域类图。

☐D) 这是“迪迪打车”系统的领域类图。

2. [多选]

给类命名时, 要注意不要在类名的最后加“情况”、“信息”、“记录”、“数据”、“表”等。请问, 针对以下哪些系统的核心域建模, 类名中可以出现刚才提到的这些文字?

☐A) 雨量数据信息监测系统

☐B) DBMS (数据库管理系统)

☐C) 疫情信息情况数据查询系统

☐D) 搜索引擎

3. [单选]

关于“宝贝”和“商品”, 经过讨论和思考, 建模人员认为在目前所关注范围内这两个词可以认为相同, “商品”更适合作为分析模型中实体类的名字, 那么, 以下说法正确的是:

☐A) 实体类起名“商品”, 但如果某类涉众觉得“宝贝”更顺眼, 和他交互的界面上依然可以写“宝贝”。

☐B) 所开发系统的所有成分应一律使用“商品”, 以建立“通用语言”。

☐C) 实体类一律使用“商品”, 和人交互的界面上一律使用“宝贝”。

☐D) 应该以“宝贝”为抓手, 将“商品”下沉到底层架构, 击穿程序员心智, 打出一套敏捷组合拳。

4. [单选]

以下给类和属性命名，最合理的是：

☐A)

菜品
- 编号
- 名称
- 做法

☐B)

菜品类
- 编号
- 名称
- 做法

☐C)

菜品
- 菜品编号
- 菜品名称
- 菜品做法

☐D)

菜品类
- ID
- 编号
- 名称
- 做法

5. [单选]

关于去除分析模型中不该有的内容，以下做法合适的是：

☐A) 需求规约（用例规约）里没提到的内容要去除，提到的内容要保留。

☐B) 观察模型中各元素的命名，是名词（包括名词化后的动词）的保留，其他去除。

☐C) 观察模型中各元素的命名，涉及计算机、软件、网络术语的内容要去除。

☐D) 逐个思考模型中的元素，如果去除它会不会影响到功能。

6. [多选]

以餐馆的取号机为目标系统。食客输入有几人就餐，请求取号，系统打印出排号单，上面的信息有餐台类型（大中小……），排队号码。

请问：如果对目标系统使用面向对象分析，根据以上信息判断，以下选项中，属于系统的分析类且命名合适的有：

☐A) 食客

☐B) 取号界面

☐C) 取号控制

☐D) 餐台类型

☐E) 餐台

7. [多选]

软件开发人员和涉众交流某个问题时，涉众提到一个他工作中的专

业术语“差动保护”，开发人员很难理解。对此，以下说法错误的有：

☐A) 开发人员应该认真去研究相关的领域知识。

☐B) 说明“差动保护”不是双方能用于沟通的“通用语言 (Ubiquitous Language)”，开发人员应该带领相关涉众建立方便开发人员理解和沟通领域知识的“通用语言”。

☐C) 有的岗位的涉众学历较低、能力较差、知识面较窄，对领域的认识有时还不如开发人员，对这类涉众所说的，如果听不懂，可以先处理其他好理解的问题，然后有空再来处理该涉众的这个问题。

☐D) 开发人员应该用敏捷+DDD 打法，给涉众赋能，连接涉众心智，通过通用语言 (Ubiquitous Language) 拉齐水位，形成闭环。

8. [单选]

产品经理王婉菲精读《软件方法》(上)，做对书上所有题目以及所有强化自测题，所写的用例规约提交给潘老师评点也得到了好评。这天，王婉菲把一份用例规约交给架构师刘庚宏，刘庚宏做面向对象的分析时，发现得到的类图中，“人员”这个类没有“姓名”属性，以下描述最正确的可能是：

☐A) 系统目前用不到“姓名”的信息。

☐B) 用例规约不能覆盖所有需求，需要补一份软件需求规约。

☐C) “人员”有“姓名”属性是缺省的，觉得哪里缺，自己加上就可以。

☐D) 王婉菲应该在用例规约中补充相应的内容再交给刘庚宏。

9. [多选]

A 公司正在开发“新一代供应链系统”，目标组织定位为动力电池厂商 B 公司，以下可以算作领域专家的有：

☐A) A 公司架构师张三，曾负责公司多个供应链系统的业务流程建模、实体-关系建模。

☐B) 著名教材《供应链管理》的作者、某高校教授李四。

☐C) B 公司仓库助理王五，熟练使用公司当前的供应链系统的仓储相关功能。

☐D) B 公司创始人、动力电池专家尼古拉斯·赵六。

10. [单选]

除了中国之外，世界上很多国家也有传统医学。例如，印国的阿育吠陀医学已经延续了五千年。阿育吠陀医学（以下简称印医）认为世界万物包括人类由土、水、火、气和空间五种元素组成，人体有中脉、左脉、右脉三条脉络，然后形成七个脉轮，一万两千个脉的支道。关于致病和治病，印医也有一套十分庞杂的知识体系，包括使用牛的粪尿、水蛭吸血以及念咒语等。

近年来，印医逐渐没落，印国的印医药管理局为了振兴印医，委托 A 公司开发一套名为“随身老印医”的系统推广全国。该系统打算封装印医知识体系和著名老印医的经验，让印国患者可随时享受正宗的印医服务。

针对“随身老印医”系统的分析 workflow，以下说法正确的是：

☐A) 应该用现代医学知识建立该系统的分析模型，为患者提供更好的服务。

☐B) 印医的知识体系是否科学，不影响软件开发。

☐C) A 公司开发人员可以和印医领域专家协作，用软件开发术语来帮助提升印医术语，得到通用语言。

☐D) 可以借此机会构建一套名为“印医驱动设计的敏锐建模方法”的全新软件开发方法学。

11. [多选]

以下文字中存在“形容词用作名词”的有：

☐A) 假烟假酒假朋友，假情假意假温柔。

☐B) 我怕我的眼泪，我的白发，像羞耻的笑话。

☐C) 在华丽的城市，等待醒来。

☐D) 凋零下的无暇，是收获谜底的代价。

12. [单选]

过去几年，核酸检测成为日常生活的一部分。假设某系统需要某个类来记住人们每一次核酸检测的时间、方式(咽拭子、鼻拭子、肛拭子)、结果等，以下选项中最适合作为这个类的名字的是：

☐A) 核酸检测

☐B) 核酸检测事件

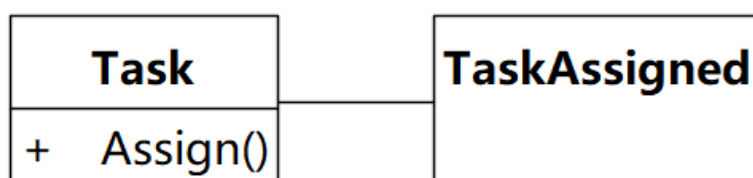
☐C) 核酸检测记录

☐D) 核酸检测单

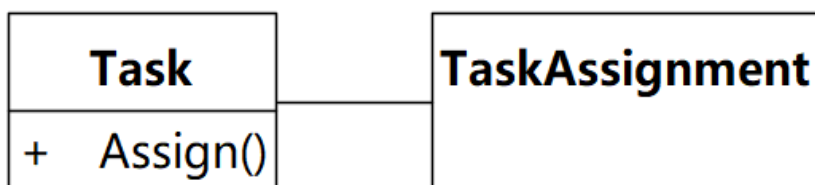
13. [多选]

关于任务和分派，以下类图中命名合适的有：

☐A)



☐B)



☐C)



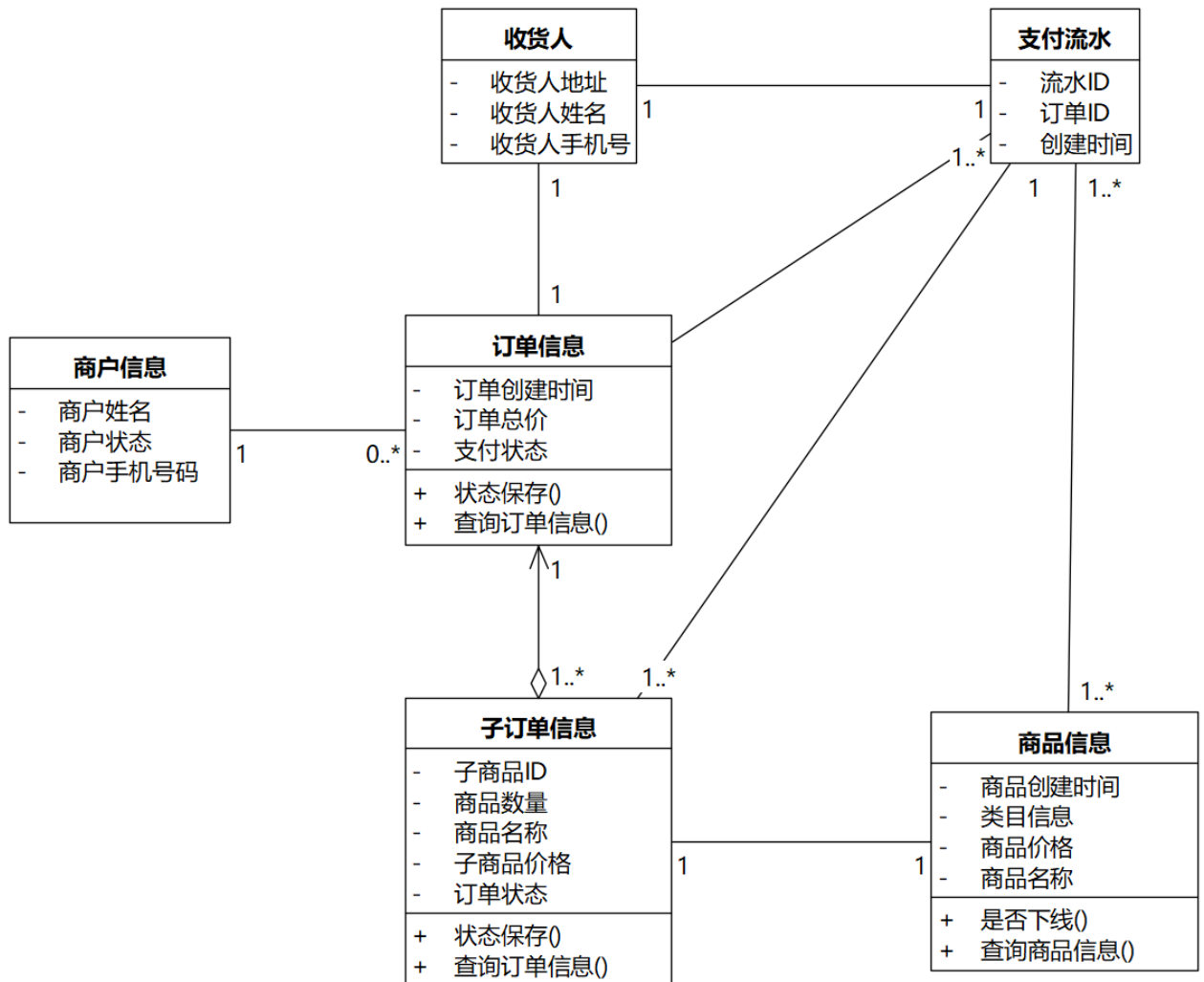
☐D)



14. [单选]

以下类图，抛开其他方面的问题不谈，只谈类名和属性名。

如果不修改类个数的情况下，删除冗余属性、类名和属性名中的冗余内容，大概有多少比例的内容可以删除？计算方法：（被删除的类名和属性名字数/图中原有的类名和属性名字数）×100%，2个英文字符算一个字。



- ☐ A) 36%
- ☐ B) 46%
- ☐ C) 56%
- ☐ D) 66%

8.2.5 审查类和属性

8.2.5.1 类名中是否有形容词

如果存在“形容词（的）名词”这样的类名，例如“待支付（的）订单”、“合适（的）会议室”，可以先把形容词从类名移除，转成类的一个状态属性。很多时候形容词和名词之间没有“的”，也要能识别出来。

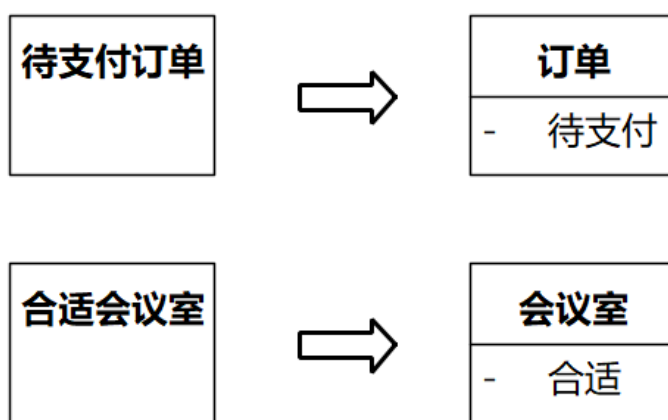


图 8-62 移除形容词为状态属性

状态属性只是临时的建模产物。在后续的建模步骤中，状态属性中包含的知识会放到更合理的位置。

“名词的名词”

有时候，名词前面的修饰词可能是名词。例如，“员工的部门”中的“员工”是名词。如果出现这样的类名，那不是什么状态属性，而是在识别类时没有识别出该有的“员工”和“部门”类。

有一些用语比较模糊，要注意理清楚真正的含义，并用更严谨的用语来表达。

例如，“会员订单”，看起来和“员工的部门”相似。经过进一步调研发现，其真实意思不是“某个会员所下的订单”，而是“下单者身份是会员的订单”。此时，“下单者身份是会员”就成了修饰“订单”的形容词，那么可以从类名删除“会员”，把“下单者身份是会员”作为“订单”的一个状态属性。当然，这个状态属性的计算可能也是通过“订单”和“会员”的关联来达到的。

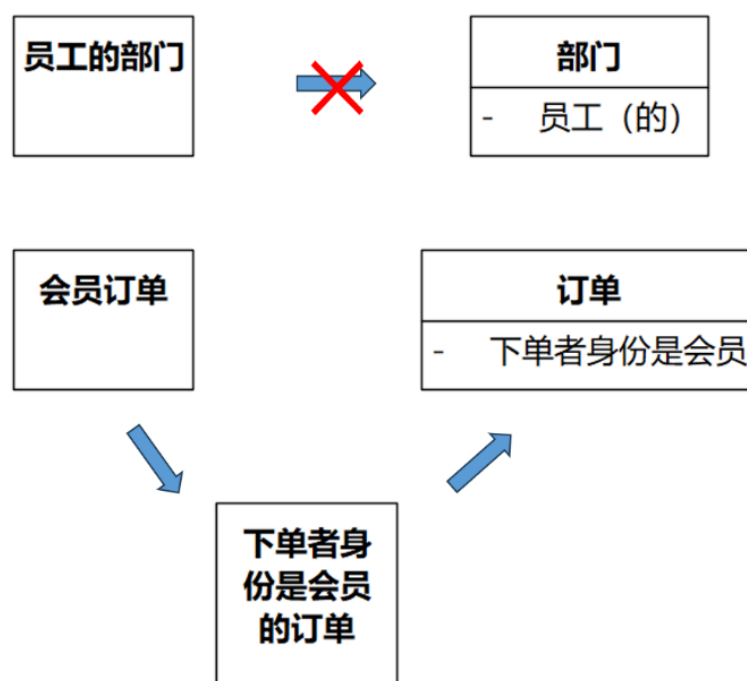


图 8-63 名词前面的名词

碰到类似的含糊用语，可以用以下方法辨别：

形容词修饰名词，相当于在该名词所代表概念的实例全集中取一个子集，它应该存在一个补集，二者的并集等于此概念的实例全集。

例如，“待支付订单”相当于在订单全集中取一个子集，同时还应该存在“非待支付（已支付）订单”——如果不存在“非待支付订单”，划分出“待支付订单”也就没有意义了。“待支付订单”和“非待支付

订单”实例集的并集即为订单实例的全集。

而“员工的部门”则不同，存在补集“非员工的部门”吗？

移除形容词的限度

图 8-62 中，“合适会议室”缩减为“会议室”，但“会议室”也可以看作“会议的室”，是否需要再缩减为“室（房间）”？

这要看系统所关注核心域知识的范围。

如果做一个“会议室管理系统”，关注到“会议室”就可以了；如果做一个“物业管理系统”，合适的类可能是“房间（室）”，“作为会议室使用”可以暂时作为“房间”的状态属性，后续步骤在处理状态属性时，可能会添加“用途”类，“会议室”是用途的一个实例。

8.2.5.2 属性是否直接描述类

类和属性连在一起说“类的属性”，应该能直接说得通，否则类和属性的搭配是不合适的。这个时候应该找到或建立合适的类，把该属性移进去。

例如图 8-64，“人员的组织名称”是“人员的组织的名称”，人员→组织→名称，中间隔了个“组织”，“类的属性”不能直接说得通，需要添加一个“组织”类，把“名称”挪过去。

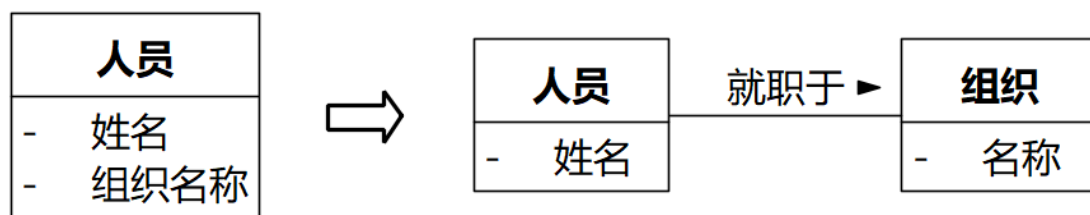


图 8-64 属性要能直接描述类

“属性要直接描述类”这个要求和关系数据库的第三范式“不存在传递的函数依赖”相似。

对于每个属性，我们都这样思考：

“类”能决定“属性”吗？如果能，是怎么决定的？如果不能，还需要什么？

--这个人的姓名叫什么？

--潘加宇

--他为什么叫“潘加宇”？

--没啥理由，他就叫这个。

“姓名”作为“人员”的属性是合适的。

★如果进一步探究，这个人为什么叫“潘加宇”也是有理由的。

理由之一可能是：“潘”是家族的姓，“加”是字辈，这由家族的规则决定。“宇”则来自鲁迅的《无题》：心事浩茫连广宇，于无声处听惊雷。了解了这个规律，还可以进一步推导，“潘加宇”如果有个弟弟，那有可能叫“潘加雷”——但这些知识，不属于系统关注的范围，所

以探究到此处就可以——“没啥理由”。

--如果知道专家是潘加宇，小时课酬能定吗？

--能，是 1500 元/小时。

--是怎么决定的？

--因为潘加宇的级别是“金牌专家”，公司规定“金牌专家”的课酬是 1500 元/小时。

决定的方式是：专家→专家级别→小时课酬。需要添加一个“专家级别”类，“小时课酬”作为它的属性。

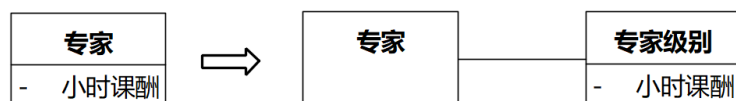


图 8-65 “类的属性” 的调整

如图 8-66 左侧，我们可以问：

--如果知道商品是“百事可乐罐装 330ml”，数量能定吗？

--不能。

--还需要什么？

--还需要知道是哪张订单。

因此需要一个类，既知道订单，又知道商品。添加“订单项”类，“数量”作为“订单项”的属性。

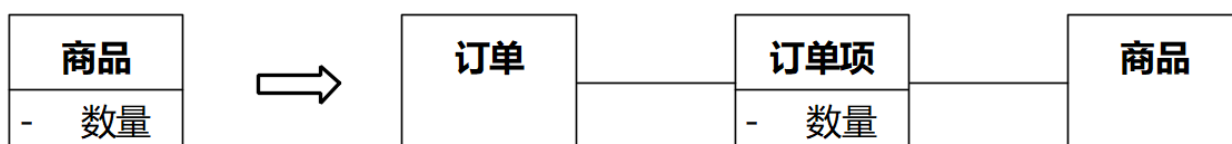


图 8-66 “类的属性” 的调整

属性如果放在了错误的类, 极有可能会导致大量不同对象的某些属性值相同, 而这反过来也可以作为线索: 当发现大量不同对象有相同属性值时, 可以检查一下, 是否有属性放在了错误的类。

就以图 8-64 为例, 如果按照左侧, 可能会有如图 8-67 的一些对象, 发现其中“人员”对象 345677 和 345679 的“组织名称”属性值相同, 都是“腾讯科技(深圳)有限公司”。这时, 可以检查一下, 是否可能漏了一个“组织”类。

345677: 人员
姓名 = 张大龙 组织名称 = 腾讯科技(深圳)有限公司

345679: 人员
姓名 = 王力红 组织名称 = 北京字节跳动科技有限公司

345679: 人员
姓名 = 罗大凤 组织名称 = 腾讯科技(深圳)有限公司

图 8-67 一些人员对象

注意, 以上是说“可以检查一下”, 不一定就需要调整。多个对象

的部分属性值相同的情况很常见，好多人姓名叫“张伟”，好多人年龄是 18 岁，并不一定要分出来另一个类。

可以合并的情况

如果 A 和 B 两个类存在 1 对 1 的关联，而且 B 只有一个属性，那么，可以把 B 合并进 A，如图 8-68。

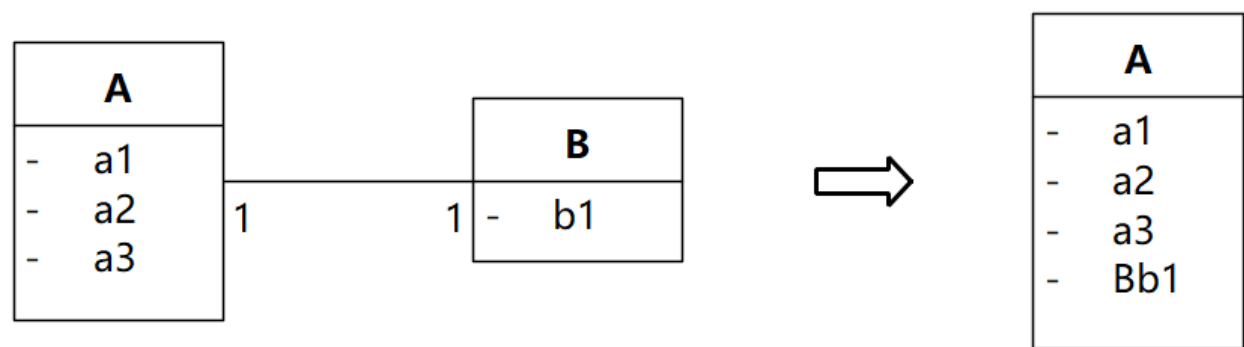


图 8-68 特定条件下合并类

例如图 8-69 的“人”和“出生”。一次“出生”事件只针对一个人，一个人只会出生一次，而且出生只关注一个属性：日期，那就可以把“出生”删掉，在“人”中放置“生日”属性。

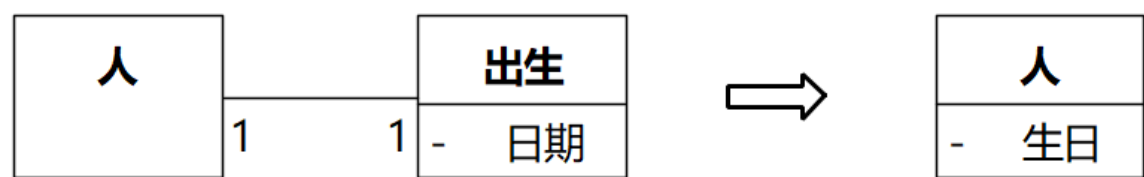


图 8-69 特定条件下合并类示例

如果一个人可以出生多次(带着前世记忆轮回,可能性趋近于 0),或者出生除了时间之外,还要关注出生医院等其他属性(可能性较大),那就不宜合并了。

还可以看出，时间类型的属性实际上是某种事件发生的属性，只不

过有的时候被合并到其他类中，看起来成为了该类的属性。如图 8-69 中，“生日”成为了“人”的属性，类似的还有：“下单时间”成为“订单”的属性等。

如果多重性不是 1 对 1，而是多对 1，那么即使 1 这一端的类只有一个属性，也不好合并。

以图 8-64 提到的“人员”和“组织”为例。

即使“组织”只关注一个属性“名称”，如果把“组织”取消，把“组织名称”合并到“人员”中，就会出现图 8-67 的情况，因为可以有很多人员关联到同一个组织。

如果组织的名称需要修改，例如“腾讯科技（深圳）有限公司”改名为“腾讯控股有限公司”，那么很多个“人员”的属性值都要修改。

同理，如果图 8-69 中的“出生”事件被多人共享，也是不适合合并的。

导致出现“属性不直接描述类”错误的原因有：

(1) 缺少抽象能力

缺少抽象能力的建模人员经常会把手上素材的信息，一一对应地映射为类和属性，导致本来属于多个类的信息被合并在一个类中。

如图 8-70，建模人员对照着一张货物运输托运单，直接把它映射成类，照抄上面的每一栏作为属性。

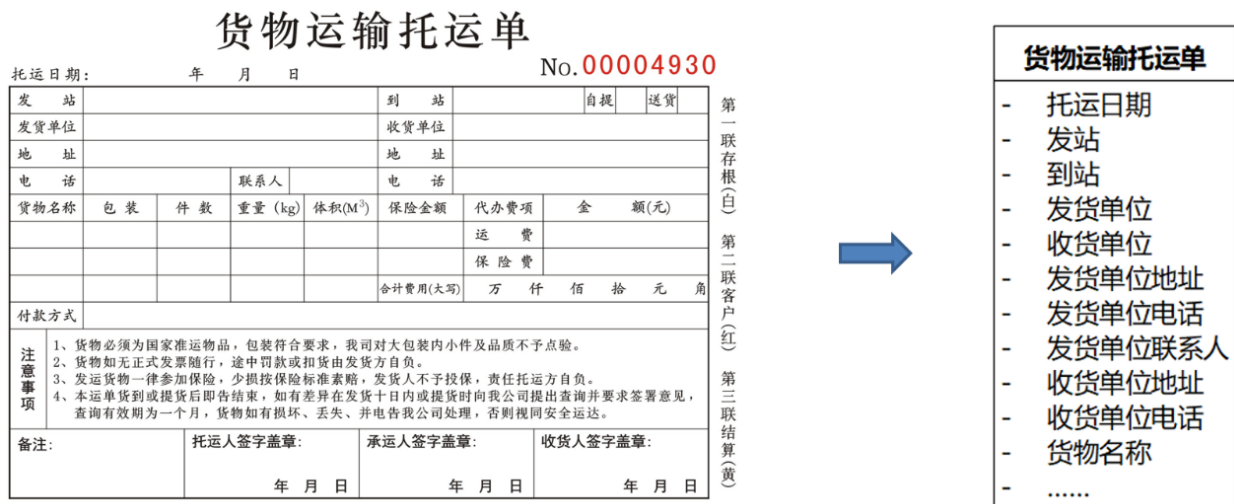


图 8-70 错误：直接把素材照搬成类

建模人员有时还觉得挺符合“类的属性”的。“货物运输托运单的货物名称”，没错啊，手边这张货物运输托运单上确实明晃晃地印有“货物名称”四个字嘛。

托运单、出库单、销售单等各种单据，以及身份证、工作证、图书卡、设备卡等各种卡片和证件，在信息时代之前就已经存在了。它们相当于某种“信息化”的存储结构，存储一个或多个概念的信息，只不过保存的载体不是电子载体，而是甲骨、铜器、竹片、木片、帛布、纸张。

现在，既然用信息系统取代了这些单据、卡片和证件，那么要建模的实体类应该是它们所存储的领域概念，而不是单据、卡片和证件本身。

图 8-70 右侧的类“货物运输托运单”应该变成如图 8-71。

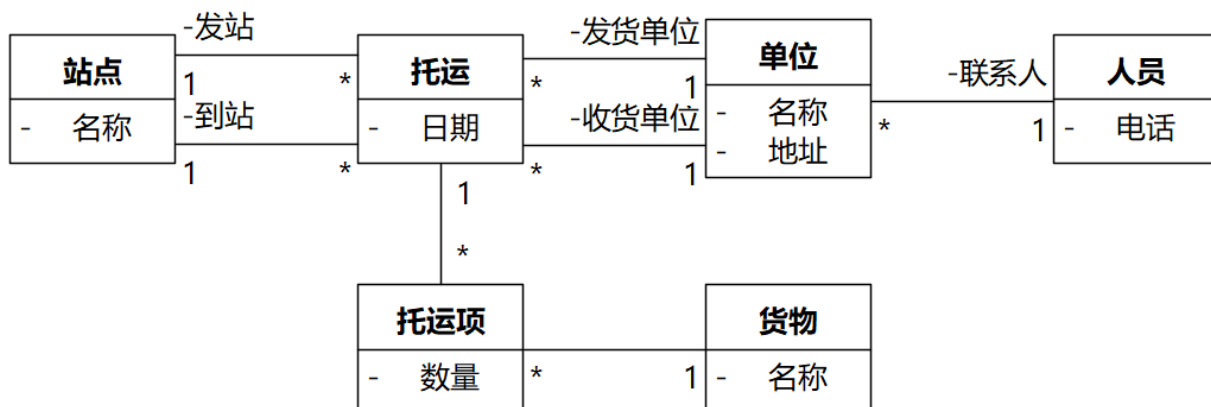


图 8-71 建模早期信息存储载体所存储的领域概念

开发团队中可能会有人唧唧歪歪，什么“性能”啦，什么“我们之前都是这样做”啦。碰到这种情况，也不用祭出鲁迅先生的“从来如此，便对么？”，你只需要问他，扔给他一张类似图 8-70 的**单，他有能力干脆利索地得到类似图 8-71 吗，**没能力就让他 TM 的闭嘴。**

★其实，如果一个人知道如何得到一个清晰、无冗余的模型，说明他具备了一定的建模能力，往往也不会在分析的时候担心这些问题，因为他知道，如果碰到性能问题，可以按照某些套路添加冗余，这些套路和目前所思考的领域知识没有关系，没有必要混杂进来。

在有能力得到图 8-71 的基础上，可以再来考虑需不需要一个如图 8-70 右侧的“货物运输托运单”类来维护当时的快照。因为随着时间的推移，对象的属性值会变化。

例如图 8-71 中，随着时间的推移，如果某个单位的名称或地址改掉了，此时从图 8-71 计算出图 8-70 左侧的托运单快照，和事件发生的当时按照图 8-70 左侧填写的信息是不一样的。

如果这样的快照很重要，可以另外加一个如图 8-70 右侧的快照类

来维护这些信息，这个类是孤立的，和图 8-71 的类不存在关联。

但这样做很容易出现沿着关联线向外延伸，雪球越滚越大的情况。如图 8-71，单位的联系人会换人，人员的电话会改变……最终可能需要一个巨大的类，把所有通过关联线连在一起的类的所有属性组合在一起，而背后的数据量也是庞大的。

更合理的做法是记录本质的变更来源。如果单位变更名称和地址是值得关注的东西，应该添加一个类记录单位变更名称和地址的细节，或者说，记录所有值得记录的对象属性值变更的细节。在此基础上，需要类似图 8-70 的某个时间点来自多个对象的属性值组合快照时，可以通过计算来还原。

在这里，我们要认清楚非常重要的一点：本质是对象之间存在关联，而不是对象属性值之间存在关联。如图 8-71，两个单位之间曾经存在的某次托运关系，并不会因为单位后来改名或改地址而变化，相对于如图 8-70 列出一堆属性值，记住“托运”和“单位”之间的关联是更本质的——当然，并不影响从本质模型还原出各种视图或报表。

正如上文提到的，如图 8-70 是在信息化时代之前的一种“信息化”的存储结构。当时不要说没有方法学，即使有方法学让你先分解概念为图 8-71，也没有计算设施把它们按需要组合成图 8-70 或更多的视图。

现在，既然有了条件，就没有必要再去模仿条件不足时的拙劣“模型”了。

很可能在这个时候，某些领域驱动设计伪创新又乘机迎合那些没有抽象能力又不愿意学习的无能之辈。伪创新的说法是：图 8-70 的快照

是发生过的事情,是不会变化的,各种所谓的“流水”才是本质!于是,无能之辈非常开心,热烈拥抱伪创新。

我们可以用科学研究类比一下:背后的规律没搞清楚时,要推测某个数据,可能是靠“经验”来推测。“经验”其实就是发生过的“流水”的记忆。当科学家研究巨量已有的“流水”(即实验数据),探索出其中的规律后,这时再推测数据,就没有必要从之前的巨量“流水”来推测了,根据归纳出来的公式推测即可。当然,之前或之后的各种“流水”可以继续保留,但它们不是本质,是现象。

何况,不是所有的系统都需要保存“流水”。电梯每天上上下下,不知发生多少次“召唤”事件,但目前的电梯系统并不会记录“召唤”事件的细节——谁召唤的、什么时候召唤的……系统只需要维护本质的行为规则,如果采用本书的方法学,可以选择用状态机来表达。

当然,也许有一天,电梯系统有了足够的存储资源,会记录所有的“召唤”流水,但这和背后的规律没有关系。系统是否记录某个事件的细节,不影响事件是否发生、事件是否产生效果,以及背后的行为规则。

(2) 受关系数据库建模的影响

建模人员有时会犯这样的错误,在一个类中放上另外一个类的属性作为“外键”。比如针对上面的例子,建模人员会想:“人员”里放“组织名称”确实不合适,但是放个“组织编码”作为外键总可以吧?其实也不可以。“组织编码”是“组织”的属性,是封装在“组织”中的秘密,“人员”不应该拥有“组织”的任何属性,它只能通过关联拥有“组织”对象,然后通过访问“组织”对象公开的操作来间接访问“组织”的属

性。

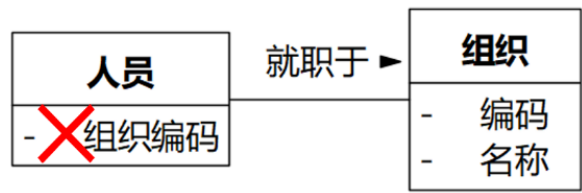


图 8-72 不需要“编码”作为“外键”

“人员”里放“组织编码”不合适，放一个无意义的标识“组织 ID”呢？同样也不可以。因为这个“组织 ID”是“组织”的标识，前文已经说了，标识属性此时不需要存在，所以“组织 ID”在“组织”里不存在，更不要说放到其他类中作为“外键”了。

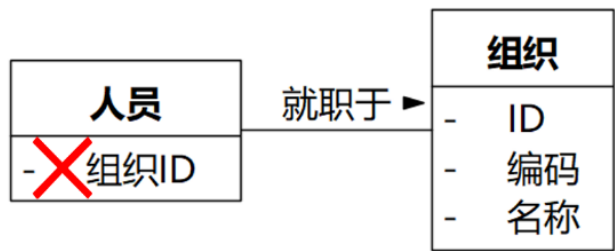


图 8-73 不需要“ID”作为“外键”

在设计工作流，需要把类图映射到关系数据库时，确实需要把“组织”表的主键（可能是“编码”也可能是生成的代理主键）放在“人员”表中作为外键，但正如上文所说，这同样是另一个领域的知识，而且映射规律和核心域知识无关。

状态属性和类的匹配

状态属性的名称是一个形容词，类型为布尔类型，用来标记对象是否处在某个状态，如图 8-74。

顾客	
-	可享受优惠

图 8-74 状态属性

这些状态属性可能是来自素材中的定语，例如，用例规约提到“可享受优惠的顾客”，那么在识别的时候可能会先把“可享受优惠”放在“顾客”类中。

和“类的属性”刚好相反，状态属性和类连在一起说，要能说得通**“属性的类”**。例如，图 8-74 中，“可享受优惠的顾客”是说得通的。

8.2.5.3 属性是否可以从其他地方推导

如果一个属性可以从其他地方推导出来，那么这个属性就是冗余的，可以删掉。

如图 8-75，人的年龄可以从出生日期计算得到，应该把年龄删掉。

人	
-	姓名
-	出生日期
-	年龄

图 8-75 年龄可以从出生日期推导

这个“其他地方”也可以是所关联的类的属性。如图 8-76，订单的总金额可以由各个订单项的金额合计得到，那么可以考虑把总金额删掉。

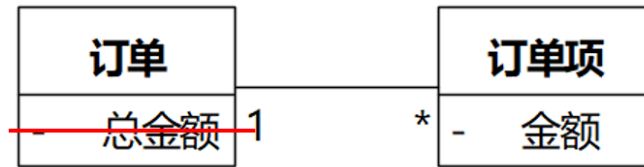


图 8-76 总金额可以从各订单项金额合计

状态属性的冗余

状态属性是冗余的,背后往往隐藏着更多的逻辑,需要进一步思考,把它变成更合适的模型内容,但这涉及到还没讲到的知识点,此处只简单举例,后文还会详述。

以图 8-62 中的“订单”为例,如果问一个“订单”对象,你是“待支付的订单”吗?如果“订单”通过自己的资源就能回答这个问题——例如,查询是否有“支付”对象和自己关联,那么“待支付”就可以作为“订单”的状态机中的一个状态。

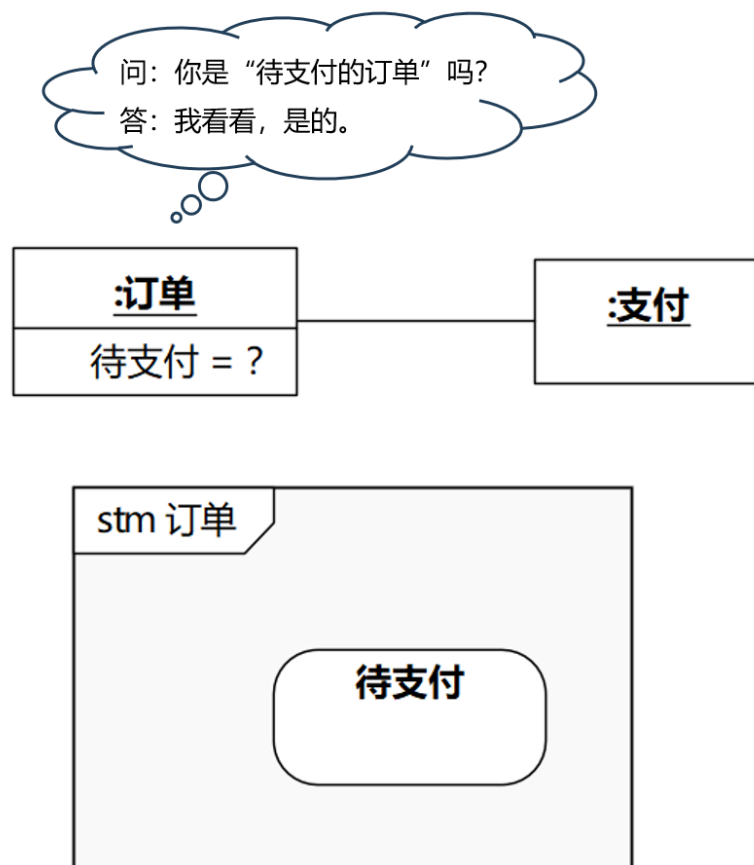


图 8-77 “待支付” 变为 “订单” 的状态

再看图 8-62 中的“会议室”。如果问一个“会议室”对象，你是“合适的会议室”吗？这时的回答是犹豫的，因为会议室合适不合适，还需要看开哪个会议，会议 A 也许不合适，但会议 B 可能就合适。这样含义的“合适”不能成为“会议室”的状态。

这个“合适”的演变可能并不简单，只是在“会议室”和“会议”之间建立一个“合适”的多对多关联，可能并不能满足要求，甚至是冗余的，因为计算哪些会议室对某个会议合适，正是系统的一个责任。如果是这样，就需要进一步建模背后的规则，如图 8-78。

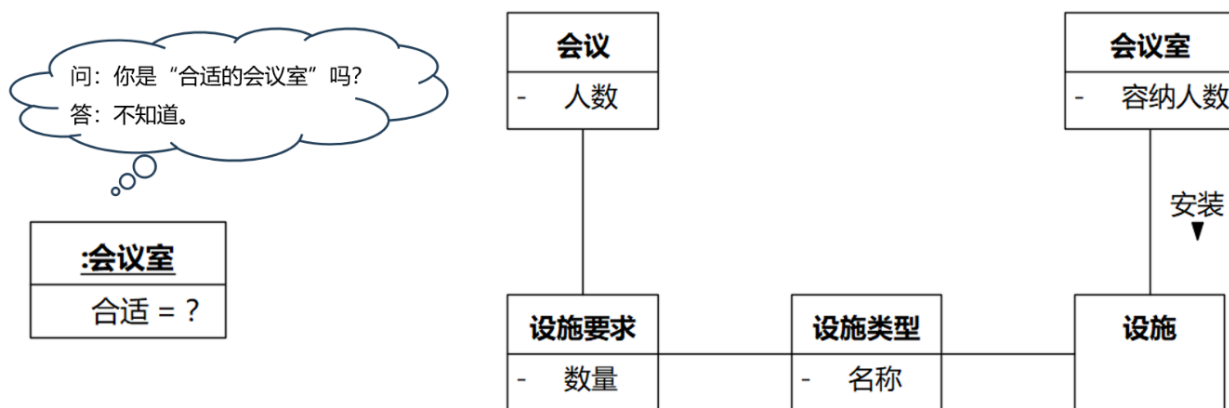


图 8-78 “合适” 背后的规则

但是，如果“合适”的含义是“当前使用会议室的会议是否合适”，有图 8-78 的加持，再加上一个“当前会议”的关联，“会议室”是可以回答这个问题的。此时，“当前会议合适”就可以作为“会议室”的状态，其他状态可能有“当前会议不合适”、“无当前会议”、“装修中”。

★把状态放在“会议”，来一个“当前会议室合适”可以吗？也可以。放在哪里更好，后文还会再探讨。

8.2.5.4 属性是否在本领域内可分解

如果属性再分解就得到另一个领域的概念，那么这个属性可以留在类中。如果属性可以继续分解成本领域的概念，那么可以考虑把这个属性独立出去变成另一个类。

如图 8-79，“人员”的“称呼”属性的类型是 String，相当于“人员”关联到 String 类。String 属于基础语义领域，已经不属于“人员”所在的“人员管理”领域，那么“称呼”可以留在“人员”中作为属性存在。



图 8-79 String 可以留在类中

而“组织”还可以像右侧所示分解成“名称”、“办公地址”等，这些概念依然属于“人员管理”领域，所以可以考虑将“组织”分离为一个类，“人员”关联到“组织”。

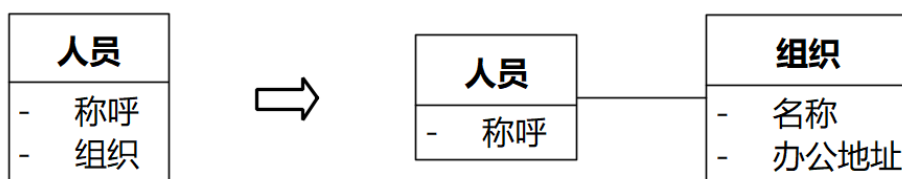


图 8-80 组织需要分离出来

注意以上的用词。我们只是说“再分解就得到另一个领域的概念”，并没有暗示这个另一个领域是什么，更没有暗示这个领域比核心域简单。任何领域，只要愿意深入研究，都是复杂的。

就拿“称呼”的类型 String 来说，String 如果用.NET 中的 String 类实现，这个类有 157 个操作，远远超过“人员管理”领域中某个类所拥有的操作数目。只不过 String 类如何构造是别人负责操心的事情，不是我们目标系统的建模人员操心的事情。

在分析 workflow，我们只需要判断某个属性再分解就到了另一个领域，或者说类型是另一个领域的类。至于属性的类型具体是哪一个类，UML 提供了一些原生类型，如果认为属性的类型刚好是这些类型之一，可以指定，否则可以先不指定，因为分析模型没有绑定到任何一个具体的编程语言和数据存储平台，而不同的编程语言和数据存储平台的类型体系

还是有差别的。

在 EA 中，把类的语言设成 none，在属性的类型选择中，就看到 UML 定义的原生类型，如图 8-81。

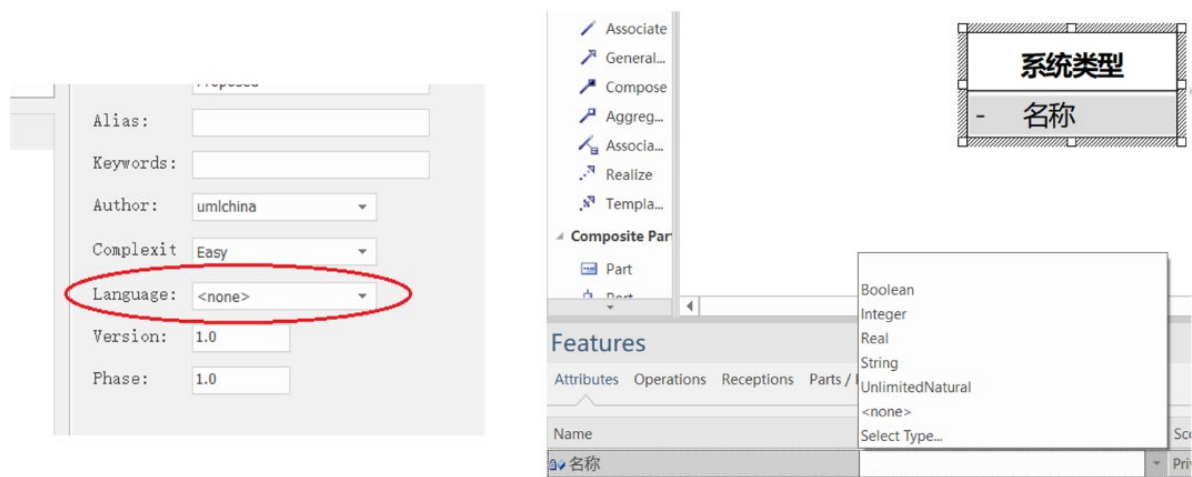


图 8-81 UML 定义的原生类型

8.2.5.5 是否有多重性大于 1 的属性

如果某个属性经过了 8.2.5.4 属性是否在本领域内可分解的检验，但该属性多重性大于 1，即所谓**多值属性**。例如，人员有多个手机号。

这时可以：

- (1) 把“手机”属性留在“人员”类中，多重性设为多，如图 8-82。

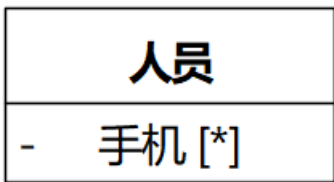


图 8-82 把属性的多重性设为多

注意，只需要说明多重性为多，不要放上一个该属性的数组或 List

之类。人员有多个手机号，这是一个领域的知识；用编程语言如何实现多值属性，是另一个领域的知识。一旦混杂，又会导致批量刷工作量。

(2) 更推荐的做法是，把该属性分离出去，如图 8-83。

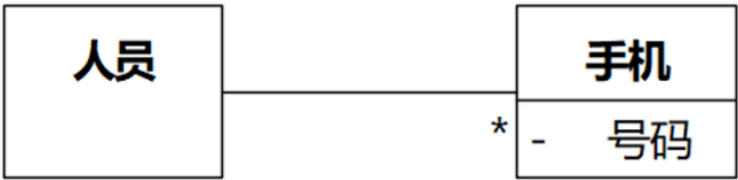


图 8-83 分离多值属性

人员有多个手机号，背后往往是有原因的。虽然现在不关注，很可能以后就会关注。例如，有的手机号是私人用的，有的手机号是办公用的，如果需要关注这些知识，那么就需要从图 8-83 转成图 8-84 中的某一个，此时只需要添加关联或者在“手机”类添加一个属性。

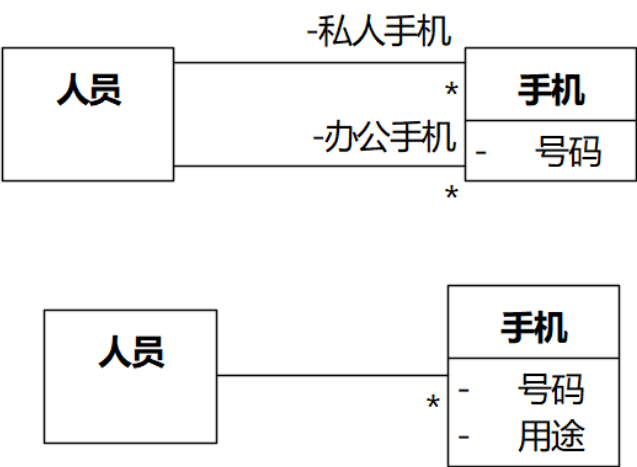


图 8-84 当需要关注更多的知识时，类图发生变化

当然，这也不是建模“人员有多个手机”的最好做法。后文我们介绍到人员相关的分析模式时，会详细描述如何建模这个领域。

以下做法是不好的：

(1) 在“人员”类中放上多个属性“手机1”、“手机2”、“手机3”……, 如图 8-85。

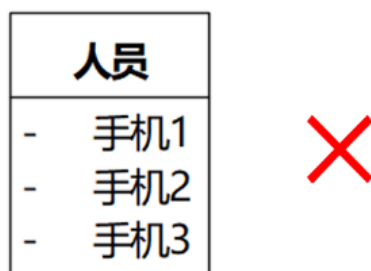


图 8-85 错误：放上多个属性

这样的做法相当于把抽象级别降到了对象级别, 或者用关系数据库建模的说法, 这是违反第一范式的。如果人员没有那么多手机号, 某些属性就会出现空值; 如果人员拥有手机号的数量大于预设的个数, 就会放不下。

(2) 只有一个“手机”属性, 属性值里用逗号分隔各个电话号码。

如果这样的做法是好的, 那不如更进一步。各个属性也不用分了, 就一个字符串。还可以再进一步, 类也不用分了, 也串在一起……持久存储或网络传输时的序列化不就是这样干的吗?

我们之所以引入各种抽象, 根源就在于软件是人做的, 人脑的容量和运行速度有限, 这些抽象可以帮助人脑应对复杂性。

8.2.5.6 属性是否对所有对象都有意义

如果通不过 **8.2.5.2 属性是否直接描述类**的检验, 那么类和属性放在一起是不合适的, 但这只是必要条件, 不是充分条件, 即使通过了也未必合适。

如图 8-86，人的姓名，人的▲▲（▲▲是男性特有的器官），人的○○（○○是女性特有的器官）好像都说得通，但如果问：是不是所有对象都应该有这个属性呢？得到的答案就不同了。

是不是有人有姓名——是。

是不是所有人都应该有姓名——是。

是不是有人有▲▲——是。

是不是所有人都应该有▲▲——不是，只有一部分人有。

是不是有人有○○——是。

是不是所有人都应该有○○——不是，只有一部分人有。

说明“人”发生了分裂，分裂成“男人”、“女人”两个子集（子类）。▲▲是男人的属性，○○是女人的属性。

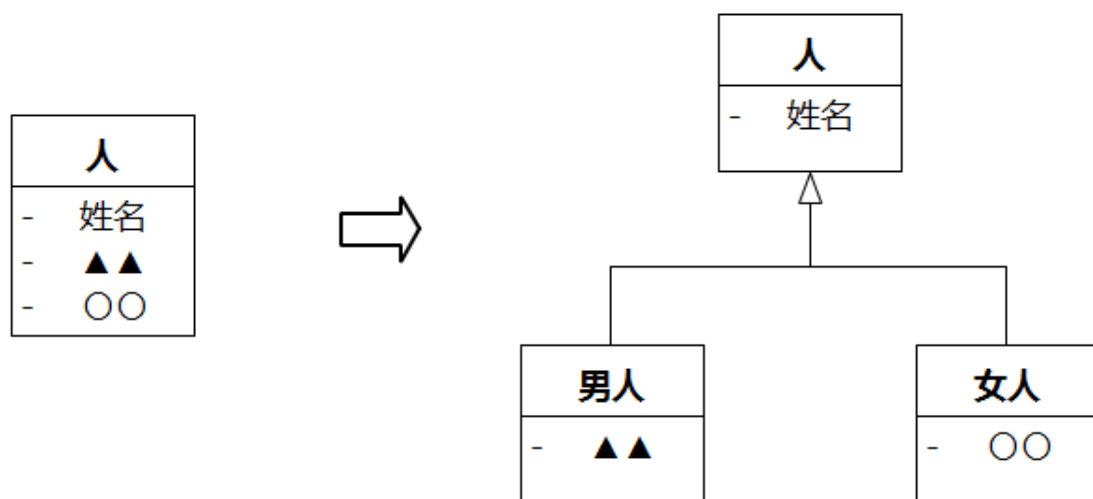


图 8-86 分解只属于部分对象的属性到子类

注意，我们说的是“应该有”，并非时时刻刻都要有。一个人刚出生，没来得及起名，这个可以理解，在后面的生命周期中，他应该会有

姓名，但一个人如果出生时没有○○，并不是还没来得及长出来。

8.2.5.7 自测题



扫 码 或 访 问

http://www.umlchina.com/book/quiz8_2_5.html 完成在线测试,做到全对以获得答案。



1. [单选]

以下符合“属性直接描述类”的是：

☐A)

电脑
- 屏幕分辨率

☐B)

会议室
- 预订人姓名

☐C)

设备
- 出厂日期

☐D)

订单
- 总金额

2. [多选]

在符合某些条件时，这样建模是可以的，请把这些条件选上。

手机
- 摄像头像素

☐A) 手机没有子类

☐B) 手机只有一个摄像头。

☐C) 摄像头只关注一个属性：像素。

☐D) 用关系数据库来保存手机对象。

3. [单选]

电子商务系统有一个类“商品”，“商品”有属性“名称”和“演示动画”。开发人员初步打算在实现时，“名称”的类型设为编程语言的String，“演示动画”的类型设为某个通用3D类库的“Animation”类。以下说法正确的是：

A 因为String比较简单，所以“名称”属性可以留在“商品”类

内。

B 因为 Animation 比较复杂, 所以“演示动画”属性应该分离出去变成关联。

C 因为 String 和 Animation 都比较复杂, 所以两个属性都应该分离出去变成关联。

D 因为 String 和 Animation 都不属于核心域概念, 所以两个属性都可以留在“商品”类内。

4. [单选]

在某个系统中有一个“电子邮箱”类, 它的对象用关系数据库存储, 放在数据库的“电子邮箱”表中。“电子邮箱”表的部分行如下:

ID	地址	用户名	密码	SMTP	POP3
11259	zhangdaong@163.com	zhangdaong	alcohol150	smtp.163.com	pop.163.com
11260	luodafeng@vip.sina.com	luodafeng	fengjie123	smtp.vip.sina.com	pop3.vip.sina.com
11261	wanglihong@163.com	wanglihong	wipeshyn1	smtp.163.com	pop.163.com
.....					

作为一名学习了本部分内容的读者, 看到这些行, 应该想到的是:

- ☐A) 类和表的结构基本一致，类的属性相当于表的列。
- ☐B) 可能需要把一些属性移到另一个类。
- ☐C) 密码不应该用明文存储。
- ☐D) 这样做不方便发挥领域驱动设计的艺术来捕获领域业务用户需求的功能架构技术逻辑规则。

5. [单选]

使用关系数据库来存储数据时，发现某个表的数据中，每一行都有某些列的值为空。从面向对象建模的角度看，这可能犯了什么错误？

- ☐A)忽略了泛化关系
- ☐B)把泛化当成关联
- ☐C)把关联当成泛化
- ☐D)把设计当成需求

8.3 建模步骤 C-2 识别类的关系

首先重复本章开头所提到的：

虽然本书先讲解“识别类和属性”，再讲解“识别类的关系”，但在实际工作中，先“识别类和属性”再“识别类的关系”这个思考顺序只是一个微小的思考周期内的顺序。建模一张类图，需要很多个思考周期。也就是说，识别类和属性→识别类的关系→识别类和属性→识别类的关系→……是交错进行的。

我们阅读用例规约或其他素材,一边思考一边建模,不管识别出类、属性还是关系,画上去就是,并不需要假装看不见类的关系,非得先把类和属性都识别完了,再来识别类的关系。

8.3.1 类的关系

8.3.1.1 泛化、关联和依赖

类的关系有三种：泛化（Generalization）、关联（Association）和依赖（Dependency），UML 表示法如图 8-87。

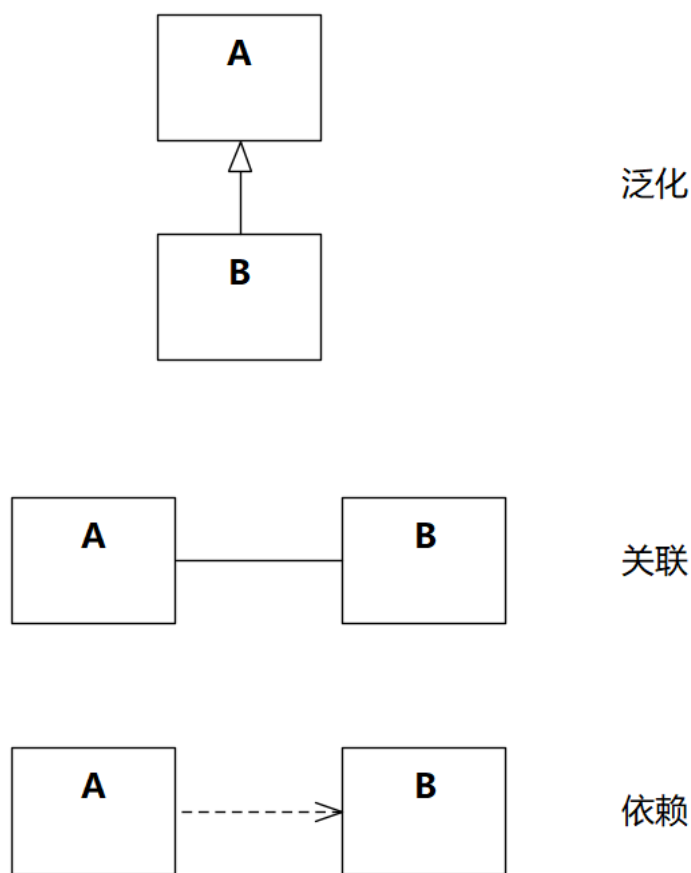


图 8-87 类的关系

泛化和关联是类的静态关系。这是系统要在一段时间内记住的关系,或者说,这两个关系属于系统要维护的“数据”。即使系统当前没有运

行需要用到这些关系的用例,这些关系依然存在,随时等待着被“使用”。

注意,此处并没有暗示“记住多久”、“如何记住”。前文说过,分析 workflows 中,任何计算所需的时间都是 0,“系统要在一段时间内记住”的这段时间大于 0 即可。就像前文所举的电梯例子,系统需要记住电梯和楼层之间的“目标楼层”关联,记住的时间可能只有几十秒。

泛化表示集合关系。两个类形成泛化,意味着超类的对象集合包含子类的对象集合。B、C 泛化到 A (或者说, B、C 继承自 A), 意味着 A 的对象集合包含 B 和 C 的对象集合, 如图 8-88。

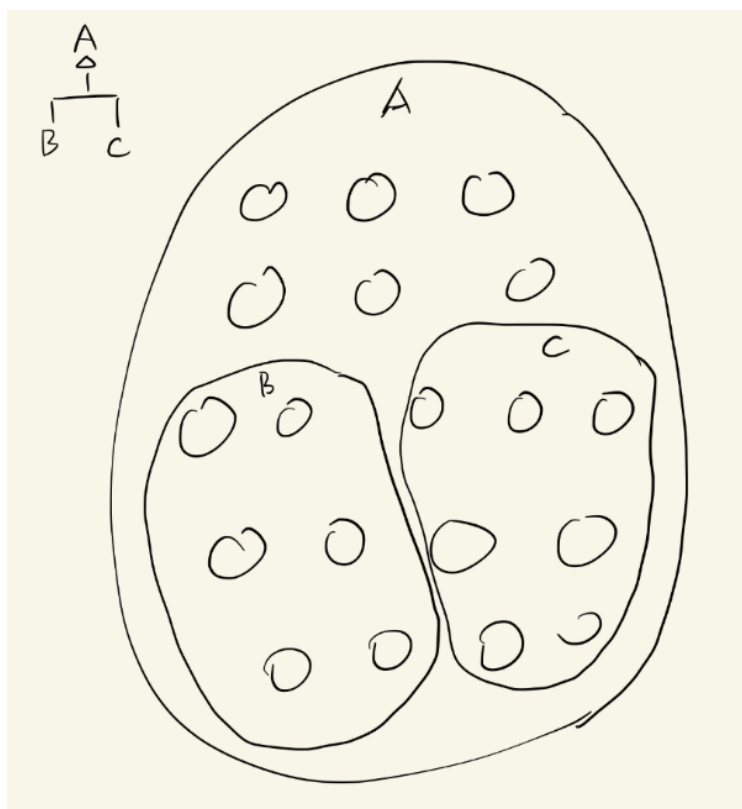


图 8-88 泛化表示集合关系

★也可以换一种说法: 子类的特征集合包含超类的特征集合。

由于是集合关系, 泛化关系是没有 1 对多、多对多等多重性的。

任何集合都是本身的子集，但此处的“包含”不考虑这种情况，也就是说，泛化的“包含”指真子集的“包含”。泛化关系只能发生在不同的类之间，不能同一个类形成自反泛化，否则就会导致批量刷废话——每个类都无条件地刷一个自反泛化。

间接的“自反泛化”也是不允许的，A 的子类或子类的子类不能成为 A 的超类。也就是说，泛化关系是传递和非对称的：对象集合 A、B、C，如果 $A \supset B$ ， $B \supset C$ ，那么可以得出 $A \supset C$ ，且 $C \not\supset B$ ， $C \not\supset A$ ， $B \not\supset A$ 。

如图 8-89 的情况，都是不允许的：

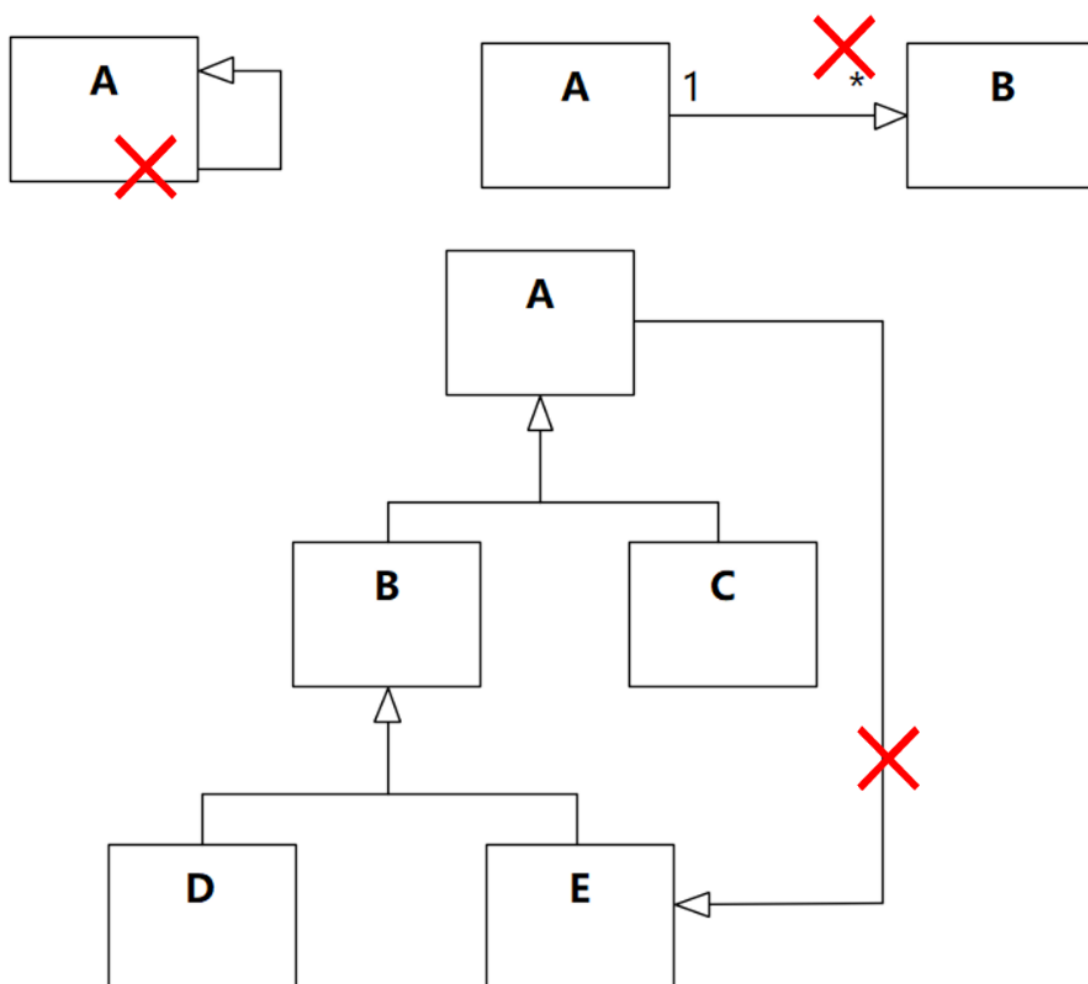


图 8-89 不允许的泛化关系

关联表示个体关系。关联可以发生在不同类之间,也可以发生在同一个类之间,意味着类的对象个体之间有关系。因为关联是类的对象个体之间的关系,所以会有 1 对多、多对多等多重性。

如图 8-90 的左侧, A 和 B、C 关联,意味着某个 A 的个体可能会和某些 B 和 C 的个体有关系;

如图 8-90 的右侧, A 自己和自己关联,意味着某个 A 的个体可能会和另外的 A 个体有关系。

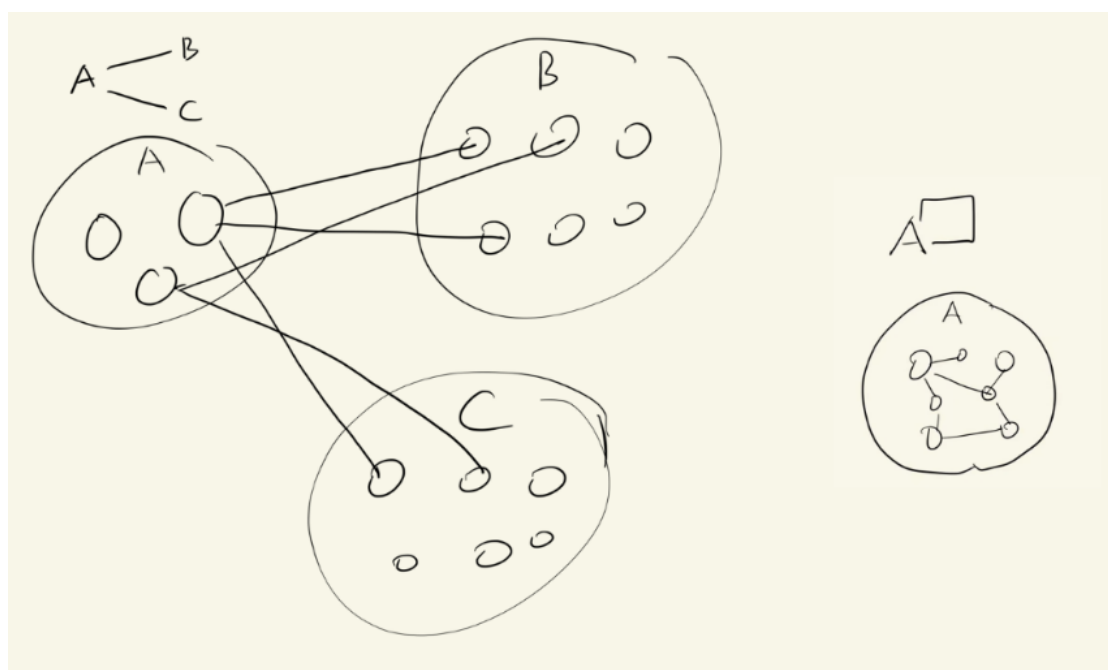


图 8-90 关联表示个体关系

是集合关系还是个体关系,这是泛化和关联的本质区别,从自然语言的表达来推断有时是不可靠的。

例如,自然语言"人有男有女"说的是泛化关系。"人有男有女"的意思不是一个人的个体里有若干男人个体和若干女人个体,而是说人的对象集合包含了男人的对象集合和女人的对象集合。

自然语言“人有手有脚”说的是关联关系。“人有手有脚”的意思不是人的对象集合包含了手、脚的对象集合，而是说一个人的个体组装了若干手和脚的个体。

读者可以自行体会一下“人有车有房”和“人有高富帅有屌丝”的区别。

对于比较熟悉的领域，例如刚才的男女、手脚，拍脑袋就可以知道是泛化还是关联，那拍脑袋就可以了。如果进入陌生的领域，回溯到集合和个体的本质区别是必要的。

泛化和关联的进一步内容，后文还会单独分节讲述。

再来看依赖关系：

如果类 B 变化,类 A 也需要变化,那么可以认为类 A 依赖于类 B。

从这个定义来看，泛化和关联也是依赖关系。泛化是子类依赖于超类，关联的依赖看关联的方向。不过，泛化和关联有另外的表示法，所以**一般说的依赖指除了泛化和关联之外的其他依赖**，例如调用、实例化等。

类之间的依赖关系用带箭头的虚线表示，线上可以标注依赖的类型(可选)，如图 8-91：

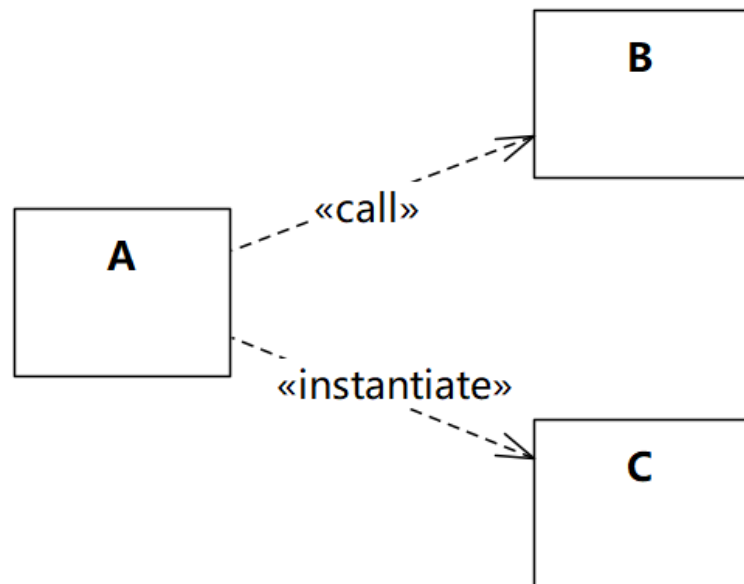


图 8-91 类之间的依赖

★显然，只有在信息系统的分析或设计模型中才会出现调用、实例化等依赖关系，纯粹描述领域知识的领域模型是不需要的。

这些依赖关系并非时刻都存在，而是在信息系统执行某个用例的某个步骤时才会产生，而且持续的时间非常短——从分析工作流的假设来说，就是趋近于 0。因此，要描述依赖关系，仅在类图上描述“A 依赖于 B”是不够的，还要具体到某个场景。

例如，要描述 A 会调用 B、C 的操作，如图 8-92 在类图上画依赖的虚线箭头，不是不可以，但还不够。

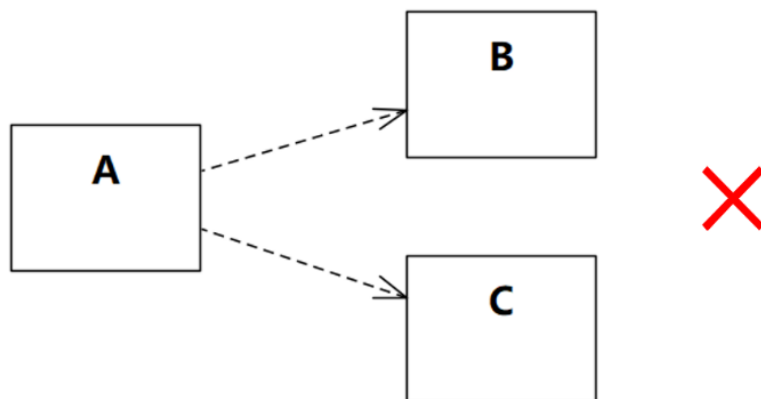


图 8-92 在类图上表达 “A 依赖于 B、C” 还不够

应该在某个用例的某张序列图（或通信图）上描述，在什么场景，进行到哪个步骤时，A 需要调用 B 的什么操作，在什么场景，进行到哪个步骤时，A 需要调用 C 的什么操作，如图 8-93。

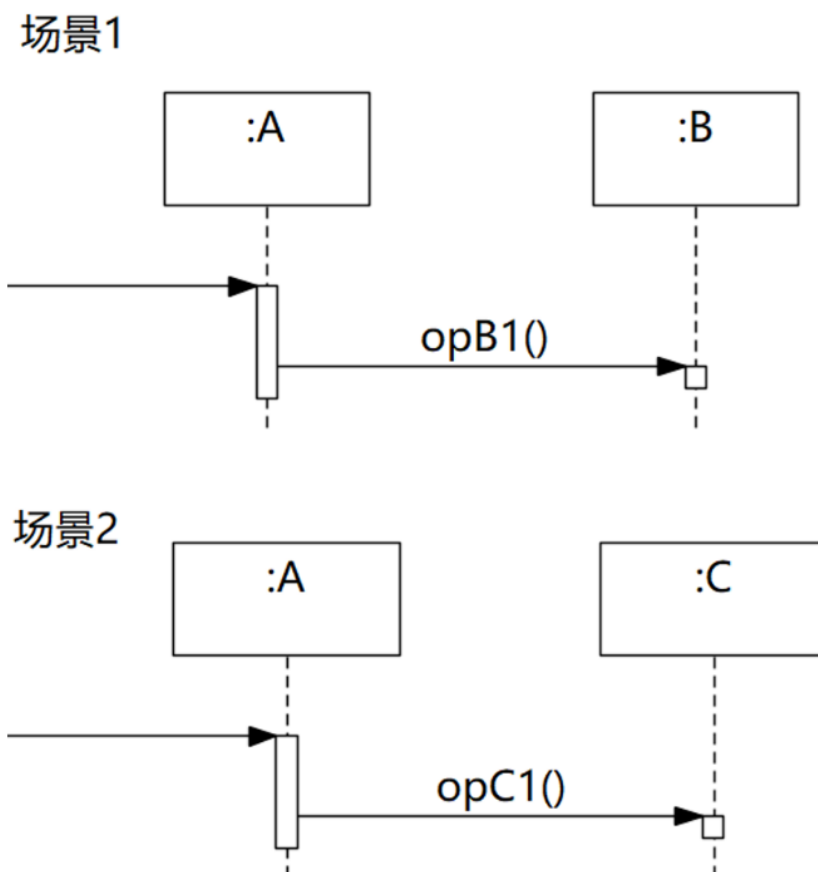


图 8-93 “A 会调用 B、C” 应在序列图上表达

有了序列图,如图 8-92 的类图上的依赖虚线就没有必要画出来了,类图上画泛化和关联关系即可。

经常看到这样的类图:上面布满了依赖的虚线,却没有泛化和关联,如图 8-94。如果这样的类图描述的是不同领域的类之间的协作,那还可以接受,如果类图上都是核心域的概念,那就要警惕了,可能建模人员根本没有去寻找泛化和关联关系。

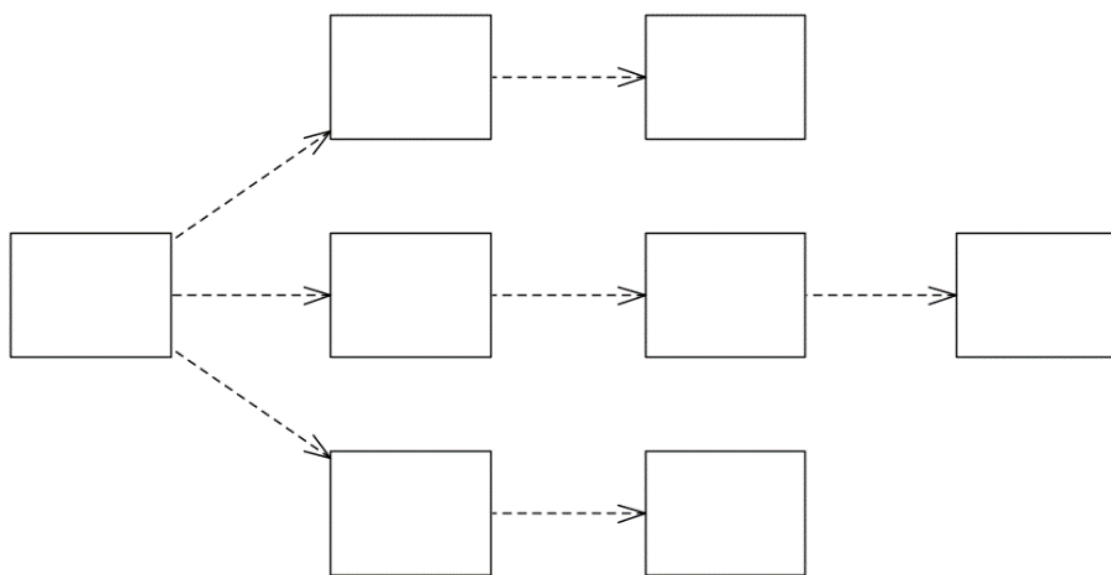


图 8-94 只有依赖关系的类图

也许有的读者会想,图 8-94 这不挺规整的嘛!问题就在于,依赖关系为什么是这个样子,很可能是没有依据的。建模人员拍脑袋定了这样的依赖关系,然后就假装自己已经“建模”了。

如图 8-95,有 ABCD 四个类,可以随意编排它们之间的依赖,不管哪一种,都可以写出代码来,编译器也不会报错。但哪一种更合理,要尽量从静态关系来找依据。

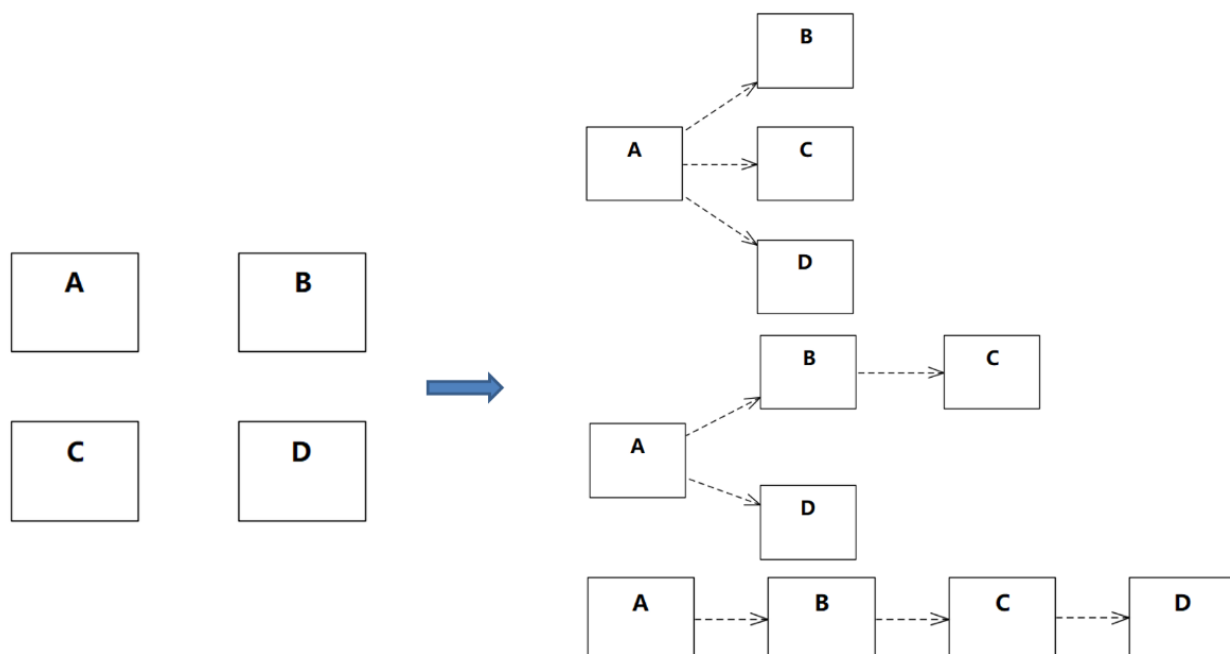


图 8-95 没有依据，依赖可以随意编排

时不时会有开发人员向我介绍他所做系统的“架构”，“您看，A 调用 B，B 调用 C……”，然后就没了。为什么要这样调用，也讲不出什么理由，反正我就是这样做了，而且做出来了，好像也能用，就行了呗，管它洪水滔天！

8.3.1.2 区分几个术语

在使用连接、关系、关联、链接等几个看起来很相似的术语时，要注意它们之间的区别：

(1) connection (连接)

普通用词，在 UML 中无特别含义。也就是说，connection 是这几个里面最垫底的、最基础的。

(2) relationship (关系)

定义为：模型元素之间的连接。注意定义中的用词是“连接

(connection)”,而不是“关联”(association),更不是“链接”(link),因为这些是在“关系 (relationship)”之后定义的。

关系的含义很广,包括类的泛化、关联、依赖,用例的扩展、包含,包的依赖、导入等等。

(3) association (关联)

定义为:实例之间存在连接的类元 (classifier) 关系。

这里的类元不一定是类,当然,最常见的关联就是类之间的关联。

(4) link (链接)

定义为:关联的实例。和对象是类的实例类似。

对比如图 8-96:

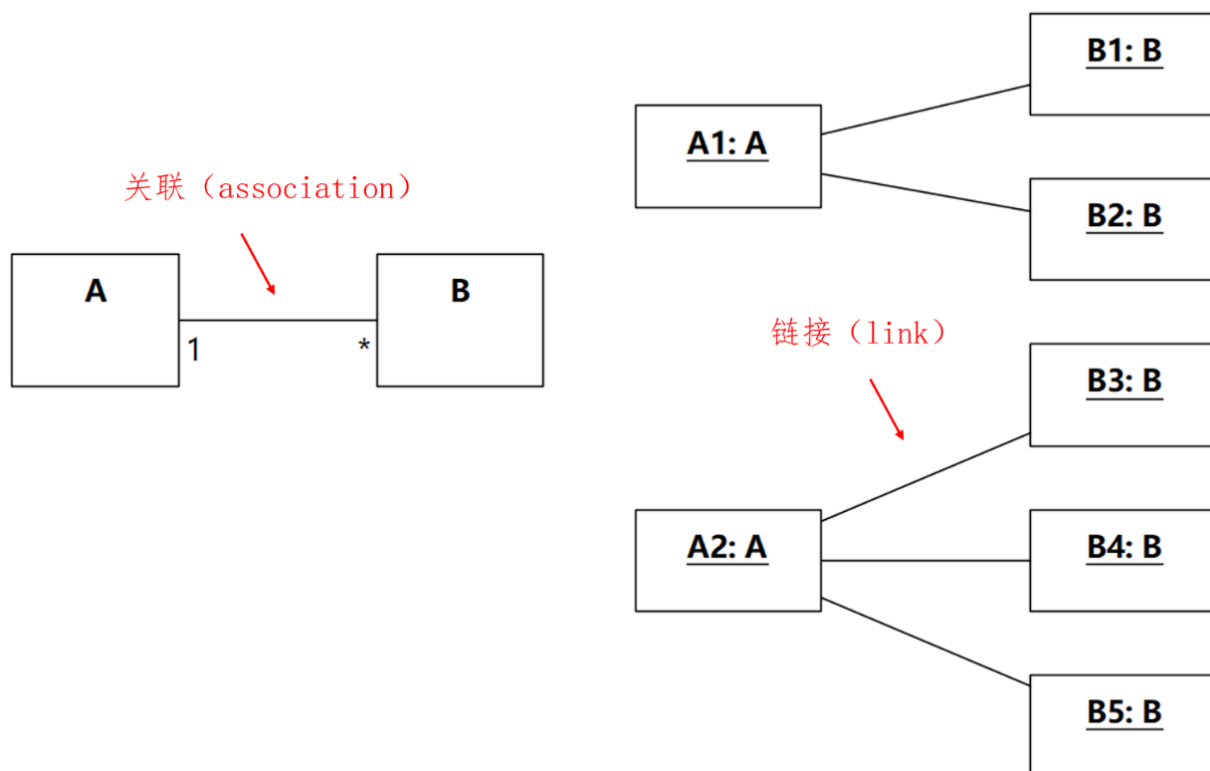


图 8-96 关联和链接 (注意, 左侧类图有多重性, 右侧对象图没有

多重性。)

8.3.1.3 推荐的关系建模顺序

根据上一小节的知识，推荐识别类的关系时，顺序如图 8-97。

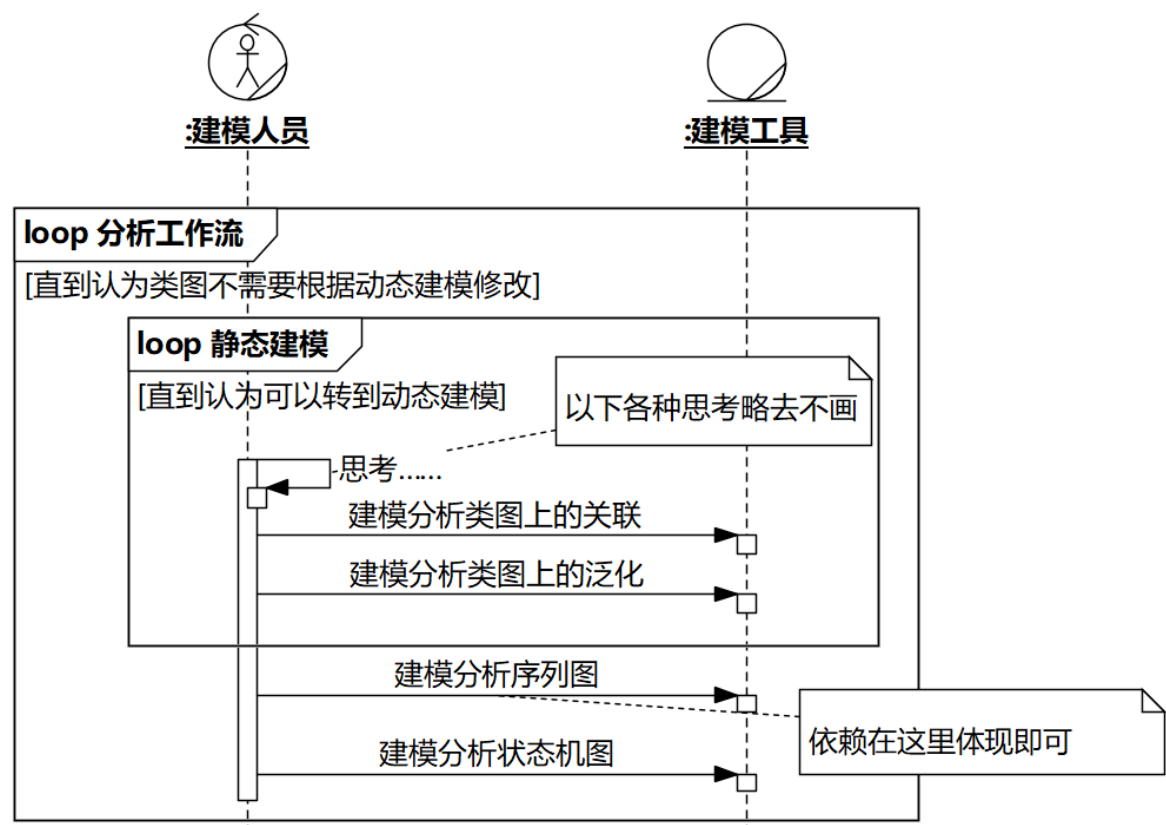


图 8-97 建模类关系的顺序

★注：

*图中的人脑思考——也就是本章的知识才是最宝贵的，但为了节省空间，就不一一画出了。

*建模类关系之前的“识别类和属性”，图中略去

*动态建模和静态建模一样，也有多个思考和建模周期，图中略去，后文再关注其中细节。

先静态，后动态

建模时，应该先在类图上建模静态关系，即泛化和关联。然后，做动态建模。通过分析序列图，把用例规约上的系统责任分配到类图上的类，并决定类之间的协作（依赖关系在这里体现即可），如有必要，还要建模状态机图。动态建模不但会给类添加行为，可能还会要求重新思考原来的静态关系。

上一段描述可以看作是识别类关系的一个建模周期，这样的建模周期在分析工作流程中可能会重复很多次。

先关联，后泛化

而在“建模静态关系”的一个小建模周期中，应该先建模关联关系，再建模泛化关系。同样，这样的小建模周期在“建模静态关系”时也可能重复多次。

每一个思考周期中的每一个思考步骤，我们都要充分利用已有的信息来思考，不要急于往下走。

8.3.2 建模关联关系

8.3.2.1 关联优先

尽量通过类之间的关联（属性可以看作和基本类型的关联）来表达核心域知识，在建模人员没有能力做到或者建模人员有能力做到但权衡利弊认为不合适的地方，再考虑引入泛化。

如图 8-98，可以如上方的模型，使用泛化关系，“男人”和“女人”

泛化到“人”；也可以如下方的模型，“人”和“性别”关联。

假设男人和女人计税的规则不同，上方的模型把变化放在“计税”操作的实现中，操作的实现里会有 0.1、0.09 等数字，而非核心域概念。而下方的模型提炼出 0.1、0.09 背后的概念“税率”，男人和女人的“计税”操作实现是一样的。

公式完全不同，而且很复杂。

严格来说，都可以用关联表达

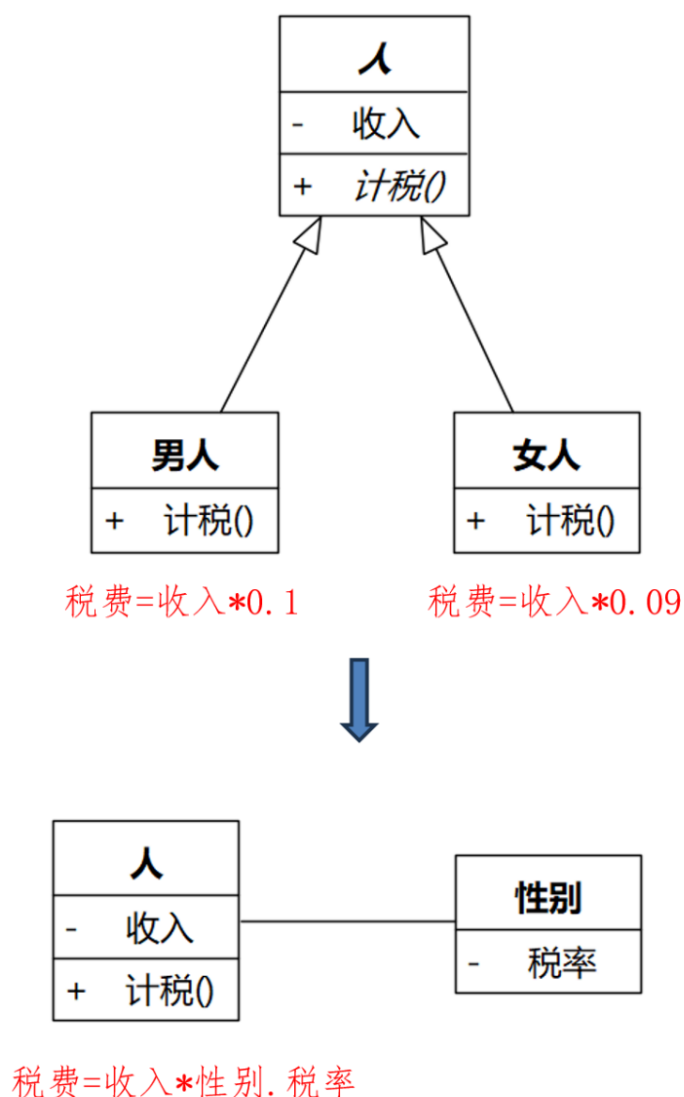


图 8-98 尽量把核心域知识通过类之间的关联来表达

★如果觉得以上例子比较偏离实际，也可以改成“结婚”操作，男人的操作实现中有“年龄<22”的内容，女人的操作实现中有“年龄<20”的内容，背后的概念是“性别.法定婚龄”。

GoF 的《设计模式》中说到：

Favor object composition over class inheritance.

优先使用对象组合而不是类继承。

意思和“关联优先”差不多，但 GoF 的《设计模式》中所说的仅仅类似于把模板方法转成策略，并没有进一步提炼核心域概念。

现实中，有的人学习了 Fowler 的《重构》，把条件语句转成泛化结构，就觉得自己“架构师”了，然后又学了“优先使用对象组合而不是类继承”，搞出若干“策略”，更觉得自己是“高级架构师”了，但这还是换汤不换药。

本书所说的“关联优先”，需要更深入的思考。

8.3.2.2 关联的 UML 知识点

图 8-99 是根据《软件方法（上）》2018 版本的用例规约得出的分析类图的一部分。

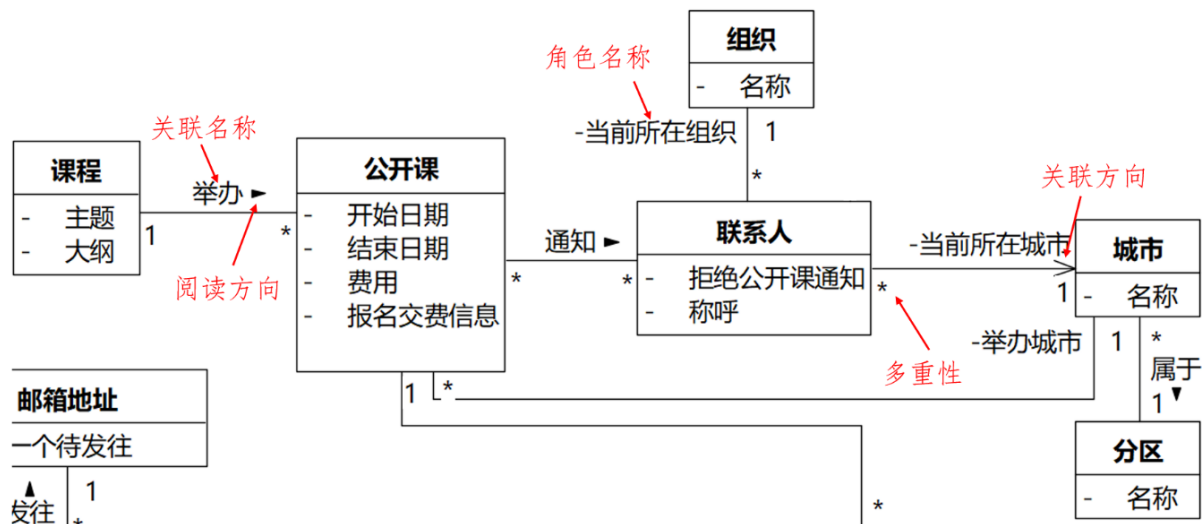


图 8-99 根据《软件方法（上）》2018 版本的用例规约得出的分析类图（部分）

图 8-99 上标注的建模元素，可以看作建模关联时所思考和表达的基本元素。其他相对“高阶”的建模元素，在讲述相应知识点时再介绍。

8.3.2.2.1 关联名和角色名

如果两个类 A 和 B 之间的关联，用“A 的 B”和“B 的 A”来称呼，就足以表达领域知识，那么可以不加关联名称和角色名称；否则，应该加上关联名称或角色名称。

如图 8-100 中，“订单的发票”、“发票的订单”可以表达领域知识，那就不用再加其他信息，但“订单的人员”会有歧义，则需要加关联名称或角色名称。如果两个类之间存在多个关联，那么关联名称或角色名称更是必须的了。

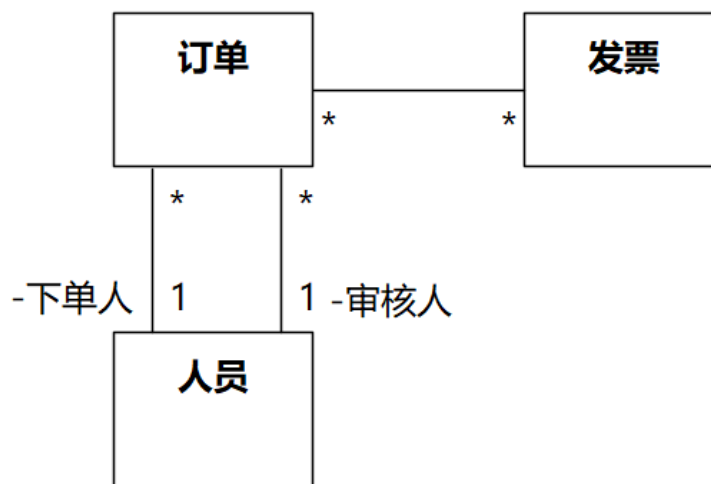


图 8-100 可以和不可以忽略关联名称或角色名称的情况

我们说的是“角色名称或关联名称”，也就是说，这两个标注一个即可。

角色名称是名词，如图 8-99 中的“当前所在组织”。映射到编程语言时，它相当于一个类在另一个类中的属性名称。角色名称是更严谨的表达，有可能的话，应该优先标注角色名称。

关联名称是动词，如图 8-99 中的“举办”。映射到编程语言时，关联名称没有意义，但在映射关系数据库时，多对多关联的关联名称可以映射为中间表的表名，有的语言（如英语）可能还需要变换一下词性。

不过在很多情况下，由于系统要关注关联的细节，多对多关联在分析工作流已经被分解为两个一对多关联，这时候原来的关联就已经变成了一个类。这个类和原来的两个类之间的关联，往往不需要额外加关联名或角色名（已有的依然保留），如图 8-101。

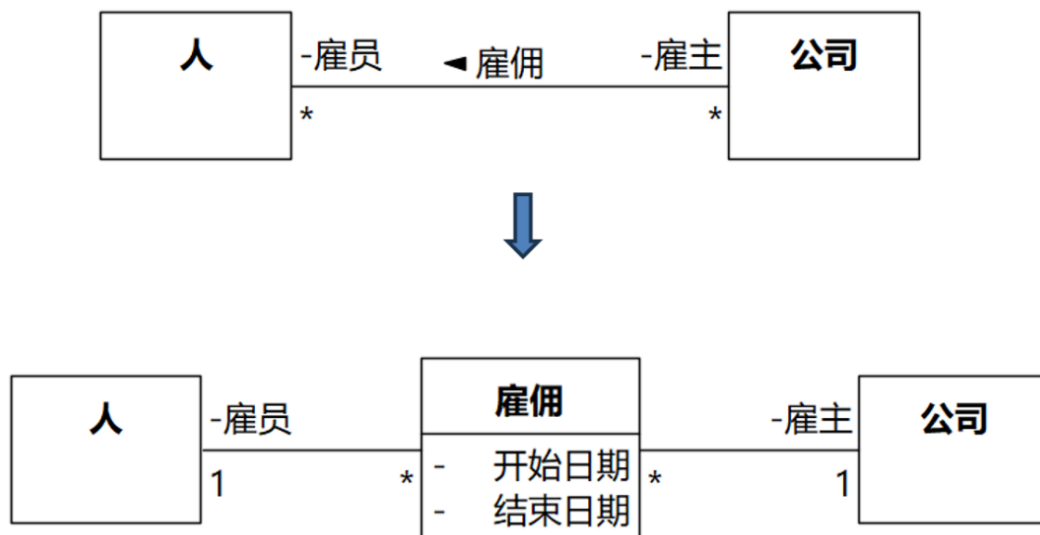


图 8-101 把关联变成类后，不需要额外加关联名或角色名

当然，如果证据暂时不足，我们可以先不标注角色名称，而是标注关联名称。

我们进一步探讨图 8-100 中“订单”和“人员”的关联。

在“人员”一端标注“下单人”、“审核人”（如图 8-102 左上），证据其实是不足的。为什么不是在“订单”一端标注“所下订单”、“所审核订单”（如图 8-102 右上）？如果暂时无法判定方向，为了严谨两边都加上（如图 8-102 左下），图上就会密密麻麻，这时还不如只写关联名称“下”、“审核”（如图 8-102 右下），等到有足够证据时，再在合适位置加上角色名称。

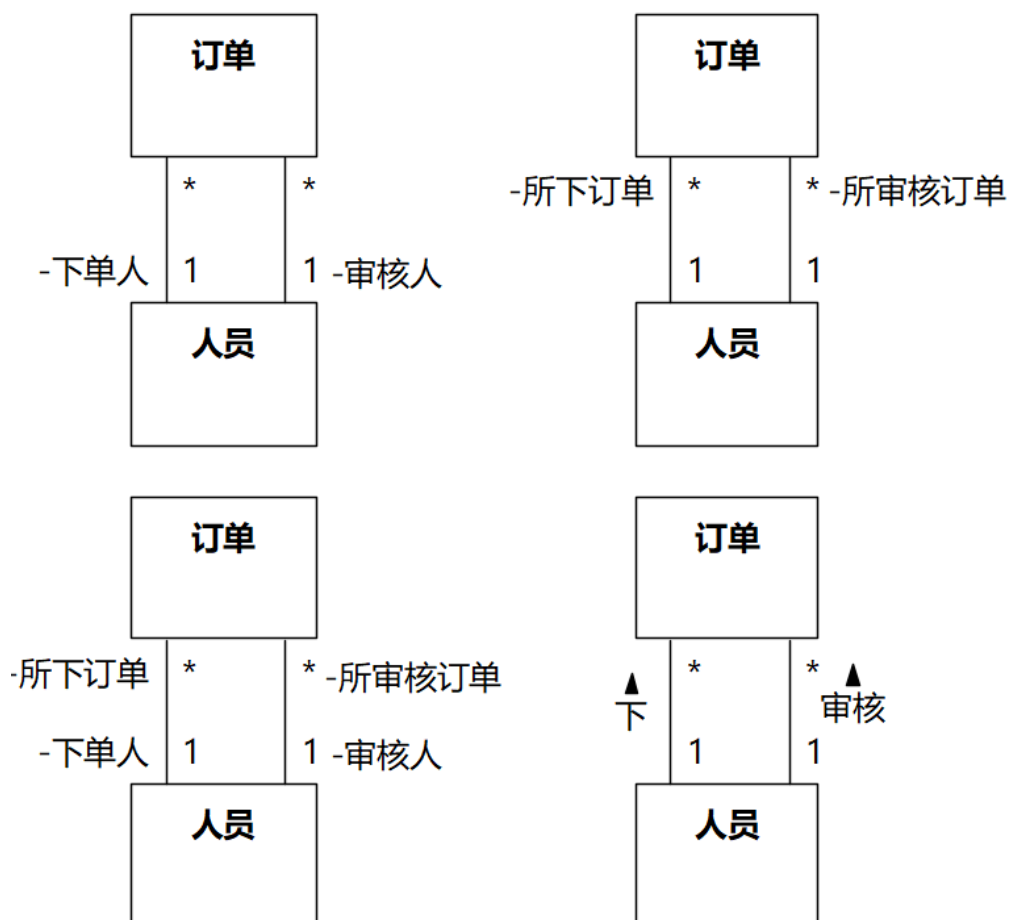


图 8-102 角色名称和关联名称的取舍

不管是标注角色名称还是关联名称，都要付出更多的思考。

有的建模人员可能并不愿意。你问他“A和B是什么关联”，他可能回答“是一个1对多的关联”，就已经觉得自己足够精细了。

有的建模人员即使命名，也是没有经过思考的废话，例如关联名称一律为“有”、“属于”，角色名称就是类名前面加一个a或m，如图8-103。如果是这样，还不如留空，等到映射到设计时再按照套路批量映射。

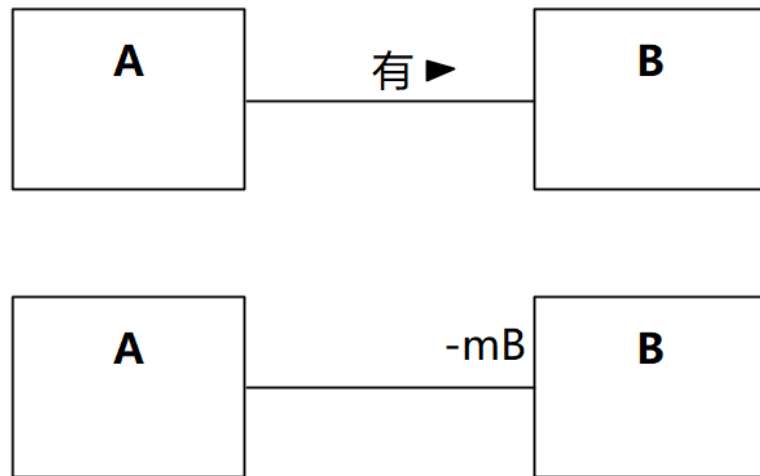


图 8-103 废话的角色名和关联名

8.3.2.2.2 多重性

首先要说一下 Multiplicity (多重性) 和 Cardinality (基数) 的区别。

多重性指关联端所允许的对象数量范围，如果数量大于 1，还包括其有序性和唯一性。基数指某个关联实例中，关联端的具体对象数量。

也就是说，多重性规定了一个范围，基数必须在这个范围之内。显然，类图上出现的应该是多重性，讨论对象和链接时，才使用基数。

★《UML 参考手册》中，cardinality 词条只是简略地说了一句，而 multiplicity 有 3 页。

常用的多重性范围表示如图 8-104。

表示	含义
1 或 1..1	1
0..1	0 到 1
*	0 到多
1..*	1 到多
具体数字如 2..8	2 到 8

图 8-104 关联多重性表示

建模中我们往往更关注的范围的上限，所以一般情况下使用 1（涵盖 0..1 和 1..1）和*（涵盖 0..*和 1..*）就好，以免图上出现大量的 0 影响阅读。等到有足够的证据说明某个关联端必须清晰标出下限，再改为更严格的上下限形式。

对于多重性,建模人员容易出现的问题是把不同时间的链接快照合并成一个。

例如，男人和女人可以有“夫妻”关联，那“夫妻”关联的多重性是多少？中国是一夫一妻制度,按道理应该 1 对 1,如图 8-105 上部。但有的建模人员会想，不对呀，有的人一生结好几次婚，那不是有几个配偶吗？是否应该如图 8-105 下部？

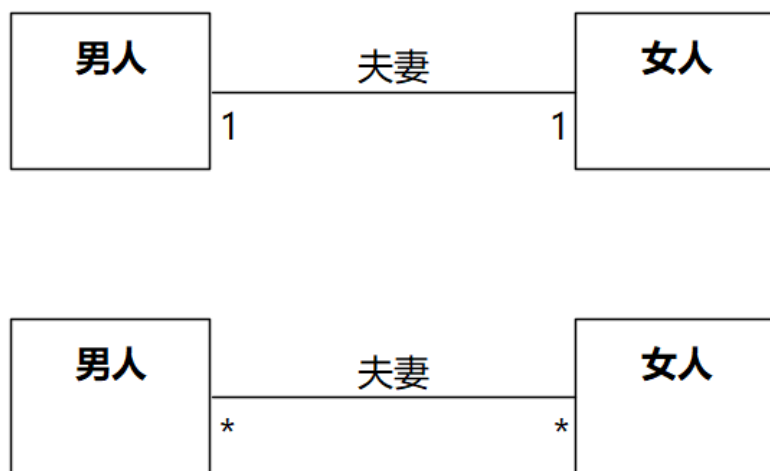


图 8-105 “夫妻” 关联的多重性

图 8-105 上部是正确的。在任何时间拍摄快照，拍到的应该是一个男人最多和一个女人有“夫妻”链接，不会拍摄到一个男人和多个女人存在“夫妻”链接，对女人来说，也是如此。

不过，拍摄快照确实可以拍到一个男人和多个女人存在“前夫妻”链接，如果要关注男人和女人的“前夫妻”关联，这个关联的多重性是多对多的，如图 8-106。

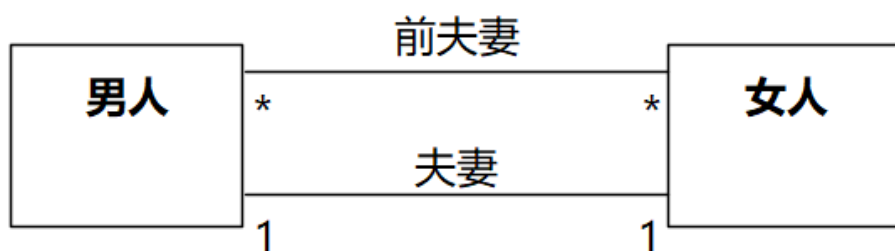


图 8-106 “前夫妻” 的多重性

8.3.2.2.3 关联的方向

如果关联不标注方向，则缺省认为是双向可导航的，如图 8-107。毕竟，双向都不可导航的“关联”没有实际意义。

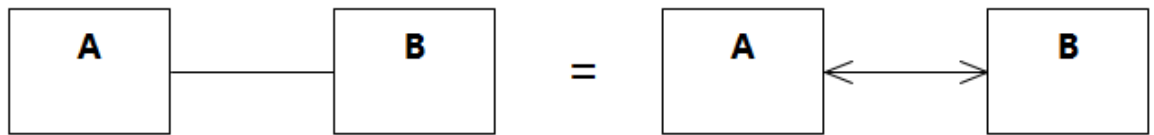


图 8-107 不标注方向，缺省认为双向可导航

如果关联标注了从一端指向另一端的方向，缺省认为没有标注的另一端是不可导航的，如图 8-108，A 指向 B，说明 A 可以导航到 B，B 不能导航到 A——当然，仅指通过这个关联不能，不代表 B 一定没有其他路径导航到 A。

同理，如果此时像图 8-107 那样，缺省认为没有标注的另一端 A 是可导航的，那图 8-108 就变成了 8-107 的右侧（双向可导航），然后等同于 8-107 的左侧，新增的方向箭头失去了意义。

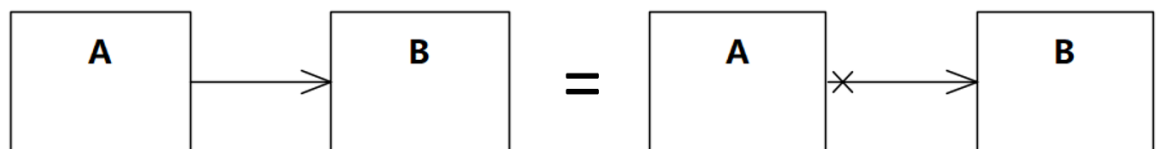


图 8-108 单向可导航

★如果要在不可导航的关联端出现图 8-108 右侧的叉叉符号，在 Enterprise Architect 中，需要右击类图的空白处，从快捷菜单选择 Properties，然后在 Connectors 页签勾选上 Show Non-Navigable Ends 复选框。

选择图 8-107 和图 8-108 的左侧来表示关联的导航方向，可以使图形更简洁。

建模的开始，如果对领域内涵把握不到位，可以暂时设为双向关联。随着对领域内涵的深入理解，会逐渐把一部分关联设为单向关联。

如何判断关联的合适方向？我们先来看看常被讨论的人和狗的例子。

日常生活中，我们可以说人养了宠物狗，也可以说狗有主人，假设目标系统要记住人和狗的关联，而且尽可能设成单向关联，那么，图 8-109 中哪一个更合适呢？

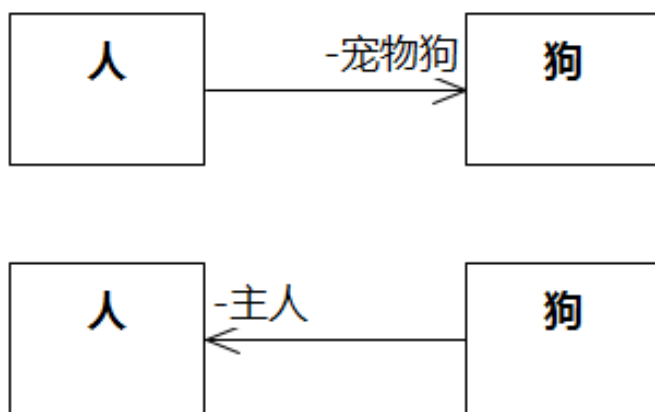


图 8-109 人和狗的单向关联，哪一个合适

人和狗之间可能一下子不好判断，两者都是胎生恒温哺乳动物，一亿年前才发生进化分叉。特别是最近一些年，由于白左思想的影响，在某些人的心中，猫狗的地位已经比人高（自己猫狗的利益甚至高于他人的生命）。

我们把问题往极端推，可以看得更清楚。如图 8-110，人有一个属性“姓名”，类型为 string（字符串），也可以看作人和 string 有一个关联，字符串的角色名是“姓名”。从平时的经验看，“人”知道 string 是合理的，但是背后的原因不是“人”比 string 更“复杂”（如“8.2.5.4 属性是否在本领域内可分解”所述），而是目标系统很可能更关注人的状态变化。

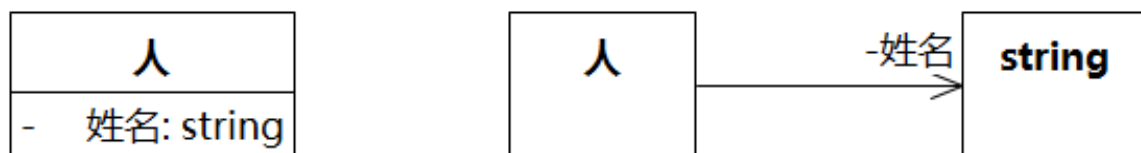


图 8-110 人和字符串的关联方向

回到图 8-109。是人知道狗还是狗知道人，判断的标准是看目标系统更有责任和有兴趣关注谁的状态。

如果目标系统更有责任和有兴趣关注“人”的状态，那么“人”应该知道“狗”；如果目标系统更有责任和有兴趣关注“狗”的状态，那么“狗”应该知道“人”。如果都有责任和有兴趣关注，那就互相知道。

不能武断地说“怎么会关注狗呢？目标系统当然更关注人的状态了！”，也许目标系统是一个狗舍管理系统，也许目标系统是一款狗的游戏——以前还有一款奇葩游戏叫《坏蟑螂（Bad Mojo）》呢，主角就是一只蟑螂。

实际上，当前的绝大多数信息系统中，和现实中有生命的“人”对应的类，与其相关的逻辑所占比例非常少。承担系统关键责任、封装复杂逻辑、我们需要为之画出复杂状态机的类，往往在现实中对应的是无生命的事物，例如“订单”、“设备”、“房间”等。

即使将来信息系统发展到更高复杂度，“人”相关的逻辑所占比例依然会很少。人类建造的信息系统，封装的是人类对宇宙万事万物（当然包括人类自身）的认识或想象，而人类自身（甚至包括其他生命体在内）的内容在这宇宙万事万物中能占多少比例呢？

另外，如果在信息系统中，人的状态机非常仔细，那就意味着人的

各种自然属性和社会属性的各种细节都被信息系统掌握,这到底是福是祸? 参见 Westworld (西部世界)、The Matrix (黑客帝国) 等影视作品。

了解了这一点,我们再来看看图 8-111 的关联,这个例子比人-狗更常见一些。

“单位”和“设备”,谁知道谁更合适? 如果目标系统是一个设备监测系统,愿景是降低设备过期漏检的比率。目标系统更有责任和有兴趣关注“设备”的状态,因此图 8-111 下部更合适。

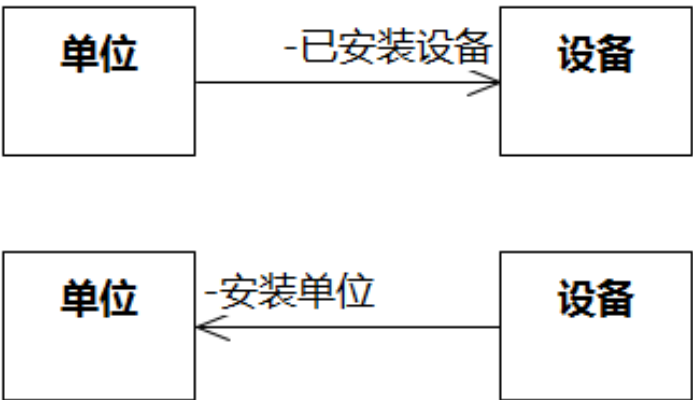


图 8-111 单位和设备的关联方向

判断关联方向时容易出现的误解:

(1) 把关系数据库中的“谁知道谁”误解为关联中的“谁知道谁”

如图 8-112, 类图中“订单”知道“订单项”:



图 8-112 类图中“订单”知道“订单项”

而在关系数据库建模中，哪个表中有另一个表的标识作为外键，判断原则是多重性。由于“订单”一端多重性为 1，“订单项”一端多重性为*，映射到关系数据库时，结果如图 8-113，却是“订单项”表中有“订单 ID”作为外键。也就是说，“订单项”知道“订单”，和类图是相反的。

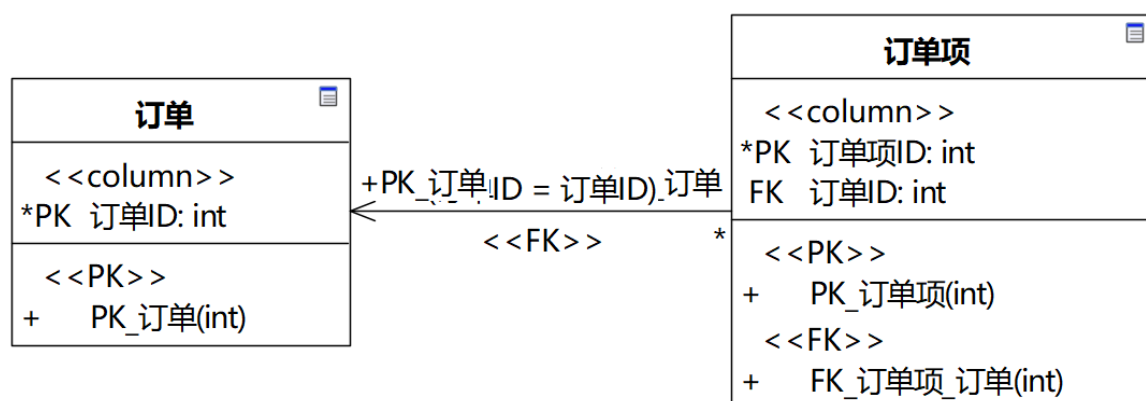


图 8-113 关系数据库中是“订单项”知道“订单”

有的情况下关系数据库和类图又是一致的。以图 8-111 的“单位”和“设备”为例，如果加上多重性，应该是一个单位安装了多台设备，映射到关系数据库的结果是“设备”表中有“单位 ID”作为外键，如图 8-114。

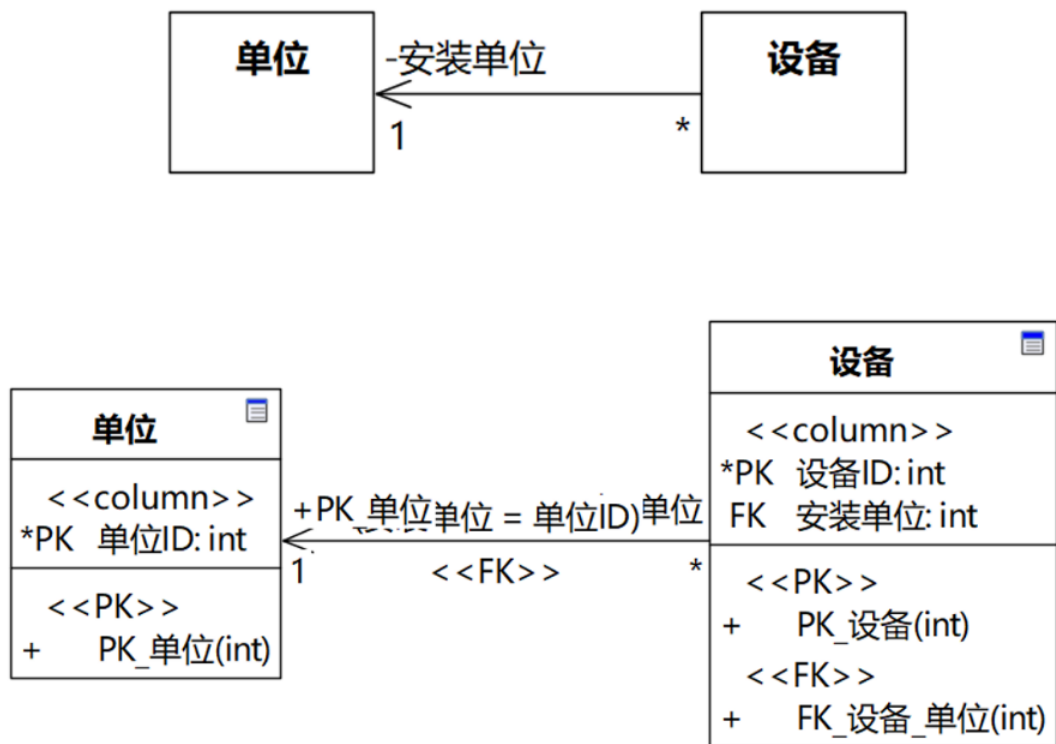


图 8-114 关系数据库和类图方向一致

(2) 把关联名称的阅读方向误解为关联的方向

图 8-99 中的“举办”、“通知”旁的实心三角形箭头，表达的是关联的阅读方向，它指示如何阅读关联才符合领域知识，并不是关联的方向。

按照图 8-99 中的箭头，领域知识为“课程举办公开课”、“公开课通知联系人”，但课程和公开课之间、联系人和城市之间谁知道谁，与这个箭头的方向无关。

如图 8-115，我们也可以把“举办”、“通知”换个词，让箭头方向调转，此时表达的领域知识为“公开课讲授课程”、“联系人被公开课通知”，仍然不影响两端的可见性。

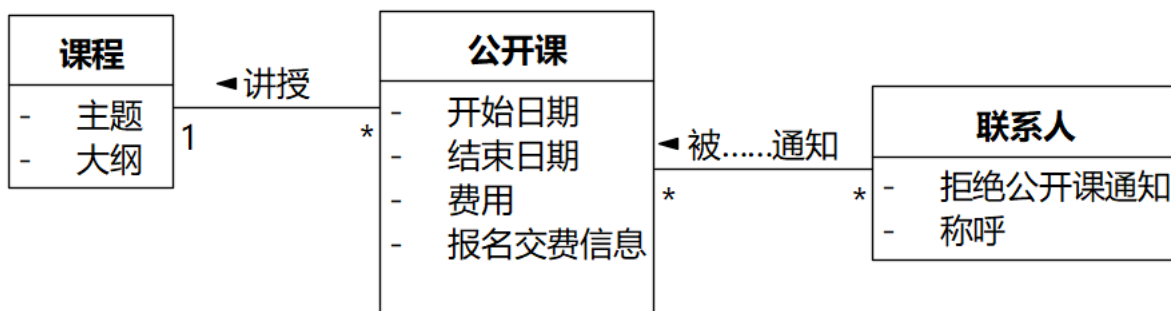


图 8-115 把“举办”、“通知”换个词，让箭头方向调转

如果是给关联命名，最好给名称添加一个阅读方向，但有的建模工具可能不支持标注关联名称的阅读方向，可以通过统一偏向源端或目标端来表达，如图 8-116 的下部。

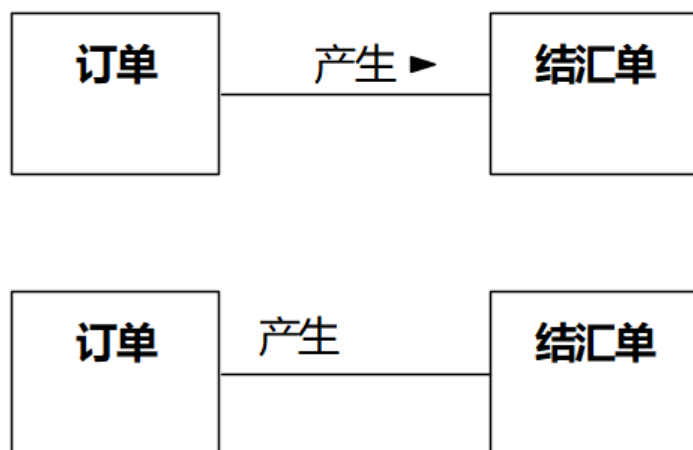


图 8-116 把关联名称统一偏向一端

如果关联端还有角色名称以及其他修饰，就会显得比较拥挤。当然，也不至于产生误读，毕竟角色名称和关联名称是容易区分的。

8.3.2.3 识别关联的思考

之前在审查类和属性时，以下审查可能已经推导出了一些关联：

8.2.5.2 属性是否直接描述类；

8.2.5.4 属性是否可以从其他地方推导；

8.2.5.5 属性是否在本领域内可分解；

在进一步识别可能存在的关联时，并没有什么妙招或捷径。

严格的做法是针对每两个类以及每个类自身，思考系统是否要维护类和类之间的关联。如果确实需要，再进一步考虑关联的多重性、关联名称、两端的角色名称。

不过这样做工作量很大。类图中有 n 个类，就需要思考 $C_n^2 + n = n(n+1)/2$ 次。后面加的 n 是自反关联的思考次数。按这样计算， $n=5$ 时，是 15 次， $n=10$ 时，就是 55 次了！

不过，绝大多数情况下不会这样做。只需要凭借对核心域知识的理解，迅速定位最应该建立的关联。如果有遗漏，后面在结合用例规约画分析序列图时也会发现。

关联不只可以发生在不同的类之间，还可以发生在同一个类上。两端都是同一个类的关联叫自反关联。

根据多重性的不同，自反关联可以表达不同的形状，如图 8-117 所示。

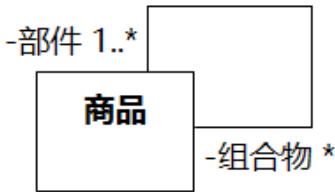
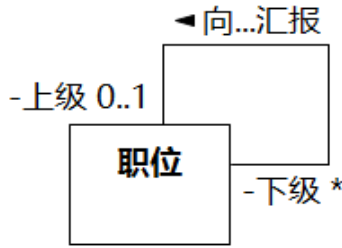
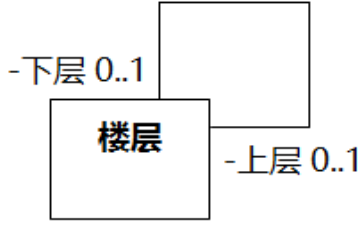
多重性	对象图形状	类图例子
对	网 	
1 对*	树 	
0..1 对 0..1	队列 	
1 对 1	环 	

图 8-117 不同多重性的自反关联。

(注意，仅有多重性并不能保证对象图形状，还需要其他的约束，这个内容后文再详谈。)

自反关联对简化模型很有帮助。在许多分析模式和设计模式中，都可以看到自反关联。例如，当类结构中的泛化层次很深时，就可以通过

把泛化结构转为更高级抽象的自反关联，如图 8-118 所示。

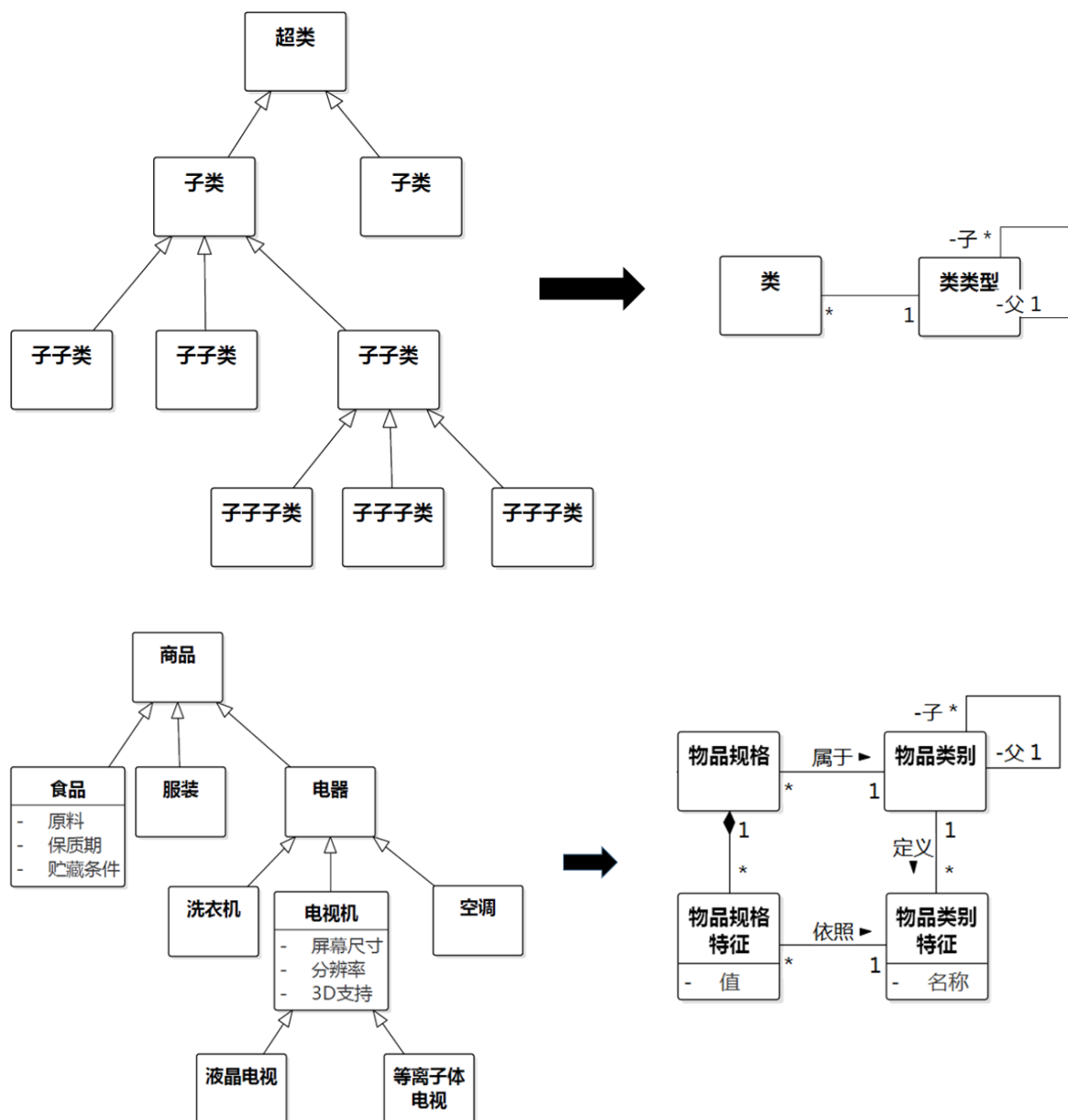


图 8-118 自反关联简化模型

8.3.3 关联的进一步讨论

8.3.3.1 关联是不得不记住的关系

前面已经说过,泛化和关联是类的静态关系,是系统要记住的关系。这个地方为什么又要强调一遍呢?

因为我们很少看见类似图 8-119 的泛化错误：

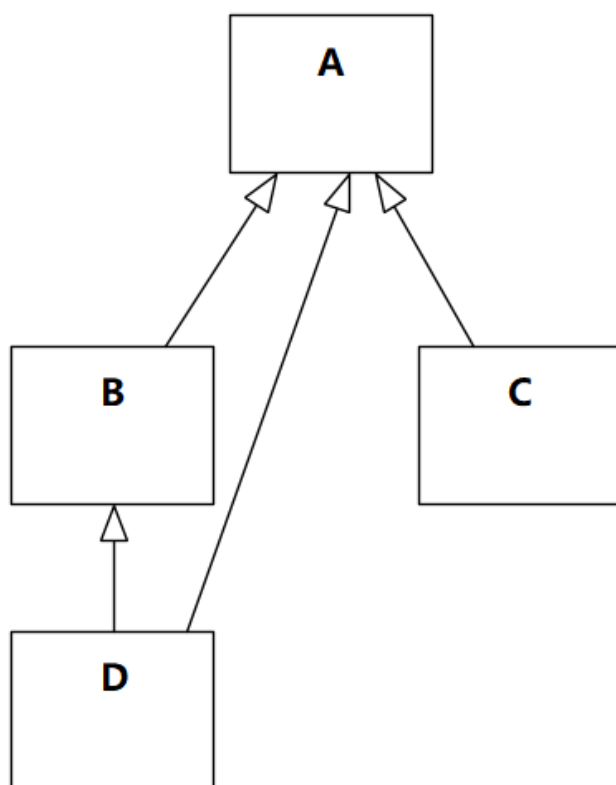
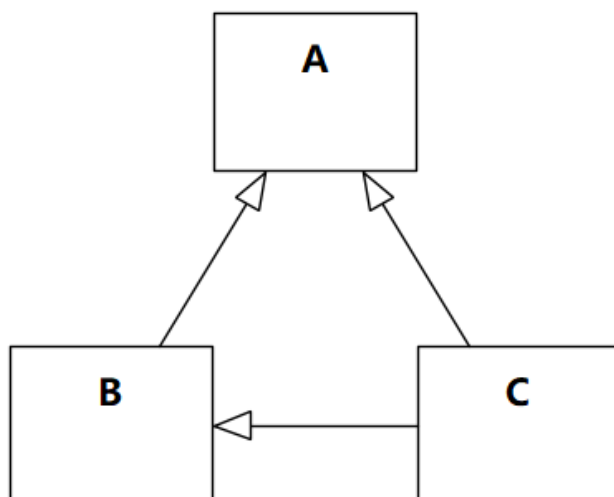


图 8-119 我们一般不会犯这样的泛化错误

我们把图 8-119 的泛化替换成关联, 得到图 8-120。建模的时候, 出现类似 8-120 的情况很多。

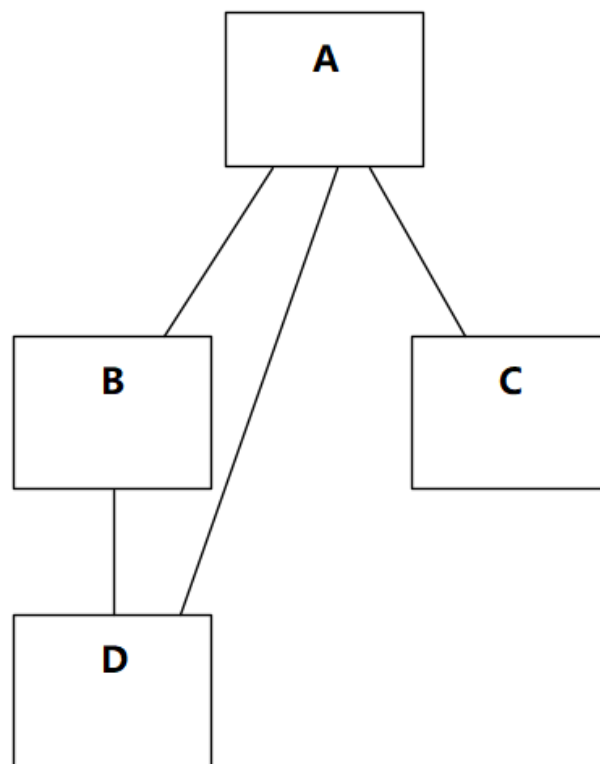
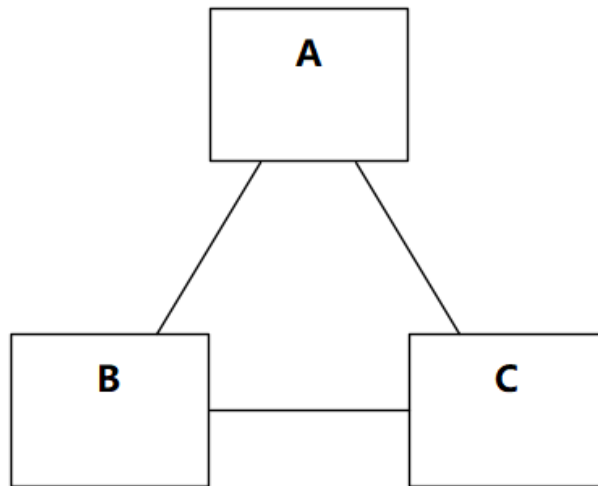


图 8-120 类似这样的关联关系很常见

图 8-119 肯定是错的，图 8-120 则有可能正确，也有可能错误。所以，我们得来看一看，需要建模的关联关系是怎样的。

假设目标系统是一个企业管理系统。

这个系统可能需要知道一个字符串是否包含另一个字符串，例如

“马宝国” 里面是否包含 “马宝” ？

是否需要像图 8-121 那样，建立字符串的自反关联？



图 8-121 字符串的自反关联

不需要，因为这个问题已经有人解决了，例如 .NET 中的 `String.Contains`。目标系统不需要一个个记住字符串之间的包含关系。

再来看 .NET 的这个 `Contains`，它是怎样实现的？

是不是通过维护类似下面的字符串之间的关联：

{ (“马宝国”， “马”), (“马宝国”， “宝”), (“马宝国”， “国”),
(“马宝国”， “马宝”), (“马宝国”， “马国”), }

然后查询这些关联得到结果？

同样不是。 `Contains` 从来没有，也不需要建立和维护这样的 “关联”，它的结果是通过已经掌握的规律计算出来的。

★.NET (或 Java) 的 `String.Contains` (背后是 `IndexOf`) 如何实现，读者可自行查询。

目标系统可能还需要计算一个数的立方，例如， $x=3$ ， x 的立方等于多少？

同理，系统也不需要记住{ (1,1), (2,8), (3,27), (4,64),} 这样的对应。

★花絮：国内的某本领域驱动设计名著说，函数式编程的加 1 函数 `add1`，是根据 `x` 的值进行模式匹配得到结果的，还给出代码：

```
def add1(x: Int): Int => x match{  
  
  case 0 => 1  
  
  case 1 => 2  
  
  case 2 => 3  
  
  case 3 => 4  
  
  //...  
  
}
```

这就相当于建立自反关联了，如图 8-122：

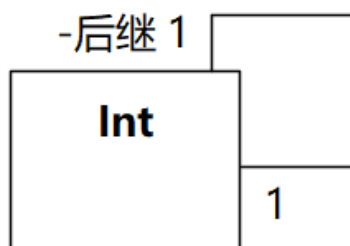


图 8-122 整数的自反关联

既然说是“模式匹配”，“模式”这个词蕴含把许许多多多个例归纳或分类的意思。模式的个数应该是少量的，例如一个枚举“商品类型”。

可是，这个 `add1` 函数，其定义域和值域都是整数，那可是一个无

限集。把每一个整数做模式匹配，岂不是有无限个模式？

我不是函数式编程专家，只是提出疑问，各位读者有空赐教。

回到正题。

目标系统可能还要知道，如果一名员工的工号是 20249527，那么他的直接上级的工号是什么？

这个就不一样了，我们能找到像下面这样的“工号→直接上级工号”的计算规律吗？

先判断工号的奇偶，如果是奇数，则乘以 3 再加 1，如果是偶数，除以 2，重复以上过程，直至得到 1。把停止后得到的数字序列按照 UTF-8 编码转成字符串，即可得到直接上级工号。

可惜，目前并没有发现这样的规律。

这时候，才需要我们建立关联来解决问题，如图 8-123。

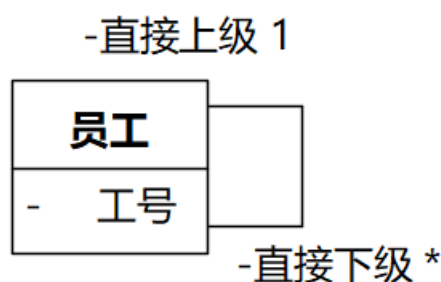


图 8-123 员工的自反关联

系统得老老实实维护“员工→员工”之间的“直接上级(直接下级)”关联，例如数据库的“员工”表有这样的一些行(假设以工号作为主键)：

工号	上级工号
20213488	20180189
20230205	20150412
20249527	20180102
.....	

图 8-124 数据库里的“员工”表

然后，目标系统在需要时查询这些数据。

我们要建模的，是当前无法计算的内容。

世界上万事万物，只要我们乐意，都可以找出它们之间的关系，但我们要建模的关联，是系统不得不一个个记住的内容，如果不记住它们，系统当前就无法完成满足需求的计算。

注意“当前”二字。

“工号→直接上级工号”到底有没有一个计算公式呢？

也许是有的，如果我们宇宙是由神级文明创造和控制的，一切运行都有其规律。如果有一天，我们掌握了其规律，很可能系统只需要记住少量参数，就可以完成任意“工号→直接上级工号”的计算。

我们用椭圆来类比。如果不了解椭圆的规律，要记住一个椭圆，需要记住很多个点的坐标值，而且得到的椭圆还不精确。后来，我们掌握了椭圆的方程，只需要记住 4 个值：两个焦点 F_1 和 F_2 ，两个轴长 a 和 b ，如图 8-125。

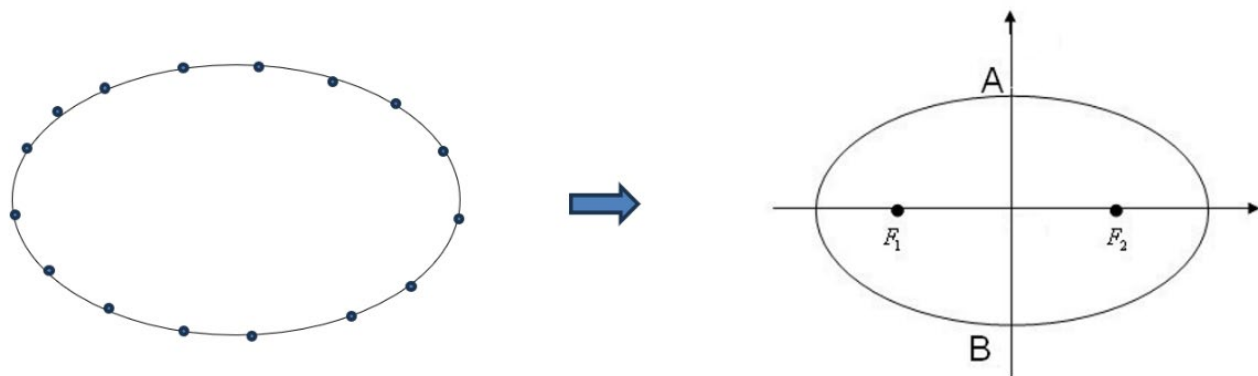


图 8-125 没掌握规律的椭圆和掌握了规律的椭圆

了解了以上的知识，我们在建模关联的时候，要学会分辨哪些是不得不记住的，哪些是可以计算得到的。

例如，我们把图 8-120 的下半部图形的内容改成一个大多数人都很熟悉的领域，并把每两个类之间的“关联”补全（共 $C(4,2) = 6$ 条边），得到图 8-126。

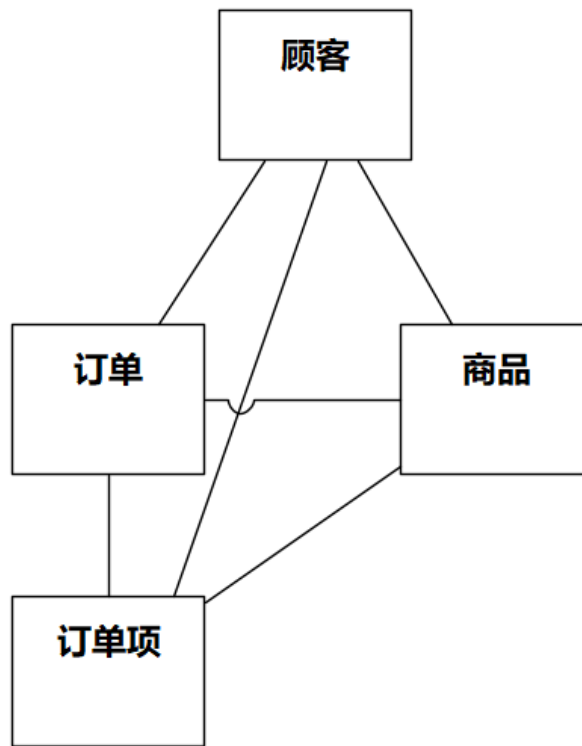


图 8-126 满满的“关联”，需要保留哪些？

删去 3 条边，只保留 3 条边就够了，结果如图 8-127：

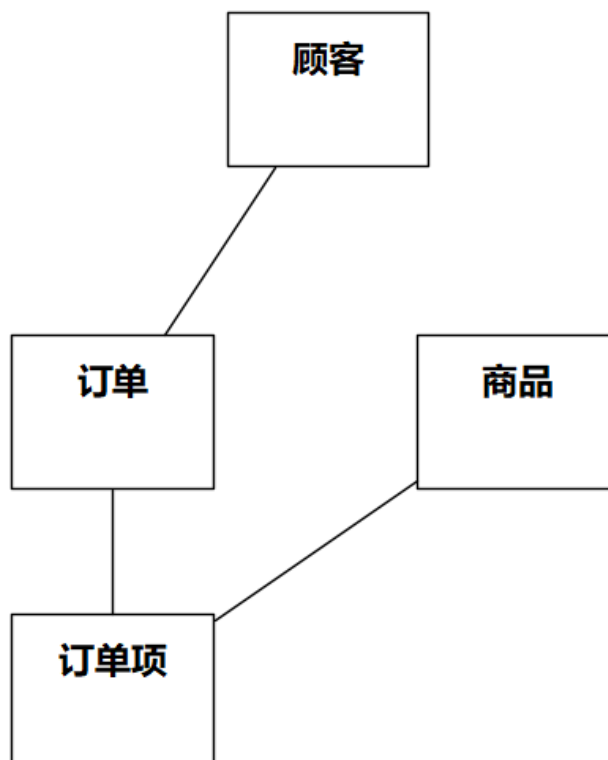


图 8-127 删去冗余“关联”后的结果

当然，这种情况下，没有能力做复杂思考只会刷废话的无能之辈可能会祭出“性能”的遮羞布，前文已讲过应对方法。

遮羞布的出现，有助于帮助我们识别无能之辈。初步识别之后，如果我们有心去翻查他的各种工件，大概率会发现各种各样的脓包。脓包在手，退可以求团结，需要亮剑时，严谨的建模匕首轻轻一划，脓包破裂飞溅满地。

8.3.3.2 关联的进一步细分

是否进一步细分各种关联，各种面向对象方法学观点不同。有的认为关联就是关联，不用再细分，有的则认为需要进一步细分。

例如，James J. Odell 就把聚合分为 6 种并详细讨论，如图 8-128。

Six Different Kinds of Aggregation
January 1994

Aggregation is a mechanism for forming an object *whole* using other objects as its *parts*. It reduces complexity by treating many objects as one object. This article examines the ways in which we form these whole-part associations by answering the following questions:

What are the primary kinds of aggregation relationships?

What kinds of relationships are often confused with aggregation relationships?

How do these various kinds of relationships help us to make correct inferences about whole-part associations?

How will these various kinds of aggregation help us during system development?

Kinds of Aggregation

Linguistics, logic, and cognitive psychology have focused on understanding the nature of relationships. One important type of relationship is the association between the parts of things and the wholes they make up. In a joint paper, Morton Winston, Roger Chaffin, and Douglas Herrmann [Winston, 1987] discussed various expressions of the whole-part association:

"The X is part of the Y," "X is partly Y," "Xs are part of Ys," "X is a part of Y," "The parts of a Y include the Xs, the Zs. . .," and similar expressions, such as the sentences: "The head is part of the body"; "Bicycles are

图 8-128 摘自 Journal Of Object-Oriented Programming Vol 5, No 8. , James J. Odell , 1994

UML 规范采取的是中间路线，把关联分为三种：普通关联、聚合 (Aggregation) 和组合 (Composition)。

用图形表示，普通关联是一根直线，聚合有一端是空心菱形，组合

有一端是实心菱形，如图 8-129。

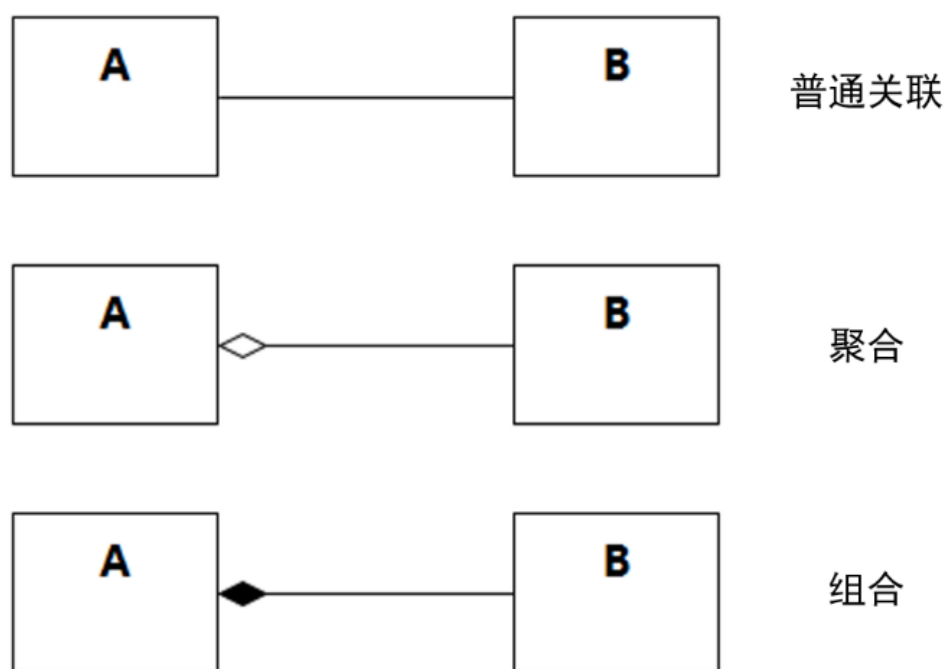


图 8-129 UML 三种关联的图示

在 UML 元模型中,把它们视为属于三个不同的 AggregationKind, 如图 8-130。

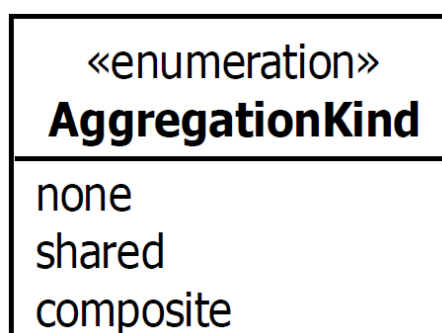


图 8-130 三个 AggregationKind

从元模型上看,“聚合”应该叫作“分享型聚合”,“组合”应该叫作“组合型聚合”,但本书还是使用“聚合”、“组合”的说法,原因后文会叙述。

聚合和组合都表示“整体-部分”关联，在类图中，菱形一端表示整体，另一端表示部分。

相对于聚合，组合还有两条额外的约束：

- (1) 在同一时刻，部分对象只属于一个整体对象；
- (2) 整体对象被销毁，部分对象也要销毁；

虽然 UML 定义了聚合的概念，但实践中要不要使用聚合，经常会引起争论。在聚合关联中，部分对象同一时刻可以被多个整体对象共享，使得“整体-部分”的概念变得模糊，和普通关联难以区分。

James Rumbaugh 等人在《UML 参考手册（第 2 版）》中认为聚合是建模的“安慰剂”。

the multiplicity, not the aggregation marker. In spite of the few semantics attached to aggregation, everybody thinks it is necessary (for different reasons). Think of it as a modeling placebo.

尽管几乎没有什么语义附加到聚合上，大家还是认为聚合这个概念是有必要的（出于不同原因）。可以把它看作建模的安慰剂。

图 8-131 摘自 Unified Modeling Language Reference Manual, 2nd Edition, James Rumbaugh, Ivar Jacobson, Grady Booch, 2004

Craig Larman 认为不需要使用聚合，在合适的情况下使用组合即可。

Guideline: Therefore, following the advice of UML creators, don't bother to use aggregation in the UML; rather, use *composition* when appropriate.

指南：因此，遵从UML创造者们的建议，不要费心去使用UML中的聚合，而是在适当的时候使用组合。

图 8-132 摘自 Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, Craig Larman, 2004

8.3.3.3 关于“整体-部分”关联

8.3.3.3.1 “整体-部分”关联的作用

之所以在关联关系中进一步划分出一个“整体-部分”关联，是希望把小的对象进一步组装成更大的对象，以获得更大的复用单元。

如果把关联定义为“整体-部分”的关联，意味着部分对象成为整体对象的部件，外部的对象不能发消息给部分对象，只能发给整体对象，再由整体对象分解和分配给组成它的部分对象，如图 8-133 所示。

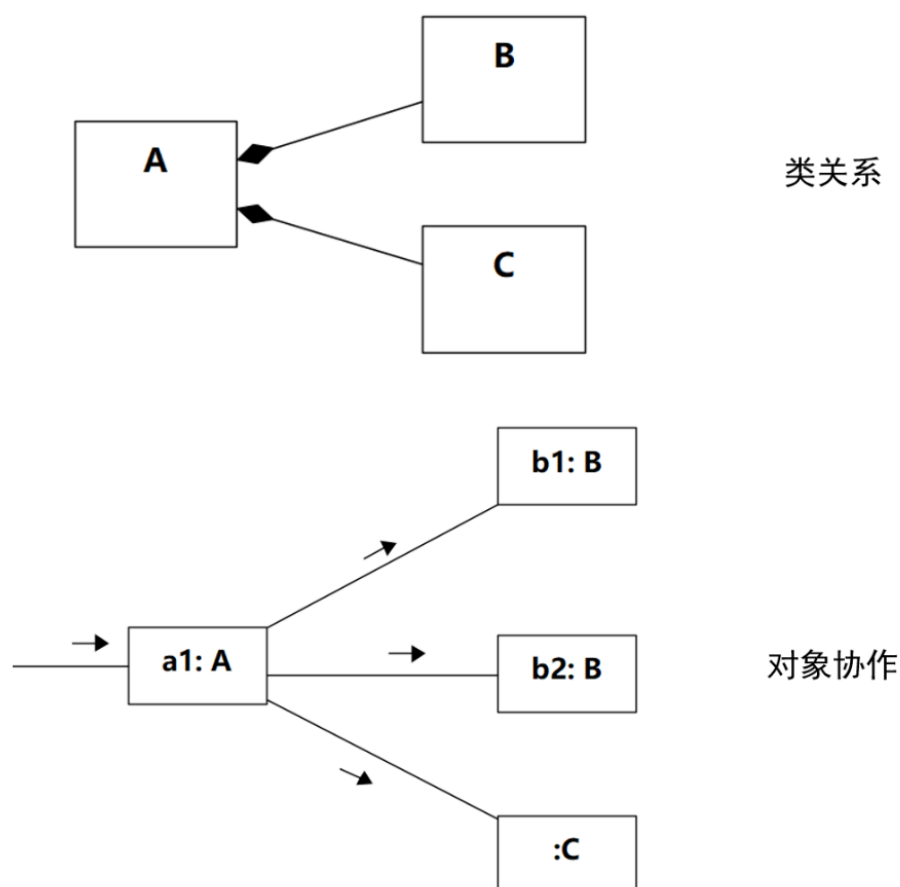


图 8-133 “整体-部分”关联影响责任分配

类图上有很多类，类之间的密切程度会有所不同。如果根据目前责任分配的情况，判断某些类之间协作的频率远超过它们和外部其他类协作的频率，而且预判将来也可能是这样，那么通过建立组合关联来强制把它们封装成一个整体来分配责任，是合算的。

而“目前责任分配的情况”也不是随意得到的，需要结合类的属性和关联来分配。这个过程会包括多次互相尝试和互相验证，详细内容在行为建模部分再讲述。

建立组合关联和公司的部门划分有类似之处。

公司不划分部门，老总一个个员工派任务也能达到目标，只是效率不高，而且不管出现什么变化都要打开老总的“代码”来修改。

划分部门之后，老总就省心多了，只需要给各部门分配大任务，部门把任务分解，再分配给部门内的各小组，各小组再把任务分解，分配给小组内的小小组.....这样，各种逻辑就会分散到各个部门、小组、小小组.....

当然，这是有代价的。划分部门之后，上级就不要越过下级去找更下级，下级也不能想找谁就找谁，都要讲基本法。

如果部门内各下级之间的协作频率远高于和其他部门协作的频率，说明这样的代价是值得付出的，部门划分以及责任的分解和分配是合理的。反之则说明不合理。

8.3.3.3.2 本书关于“整体-部分”关联的使用建议

建议一：在没有足够证据的情况下，一律使用普通关联，不用组合

(聚合) 关联。

只有经过序列图、状态机图等进一步建模核心域逻辑之后, 有足够证据支持定义为组合 (聚合) 关联更有利, 才定义组合 (聚合) 关联用于指导后续其他用例的责任分配。

经常看见有 “架构师” 随意使用组合, 图 8-134 是一张学员发给我评点的类图。可以看到, 图上到处都是菱形。

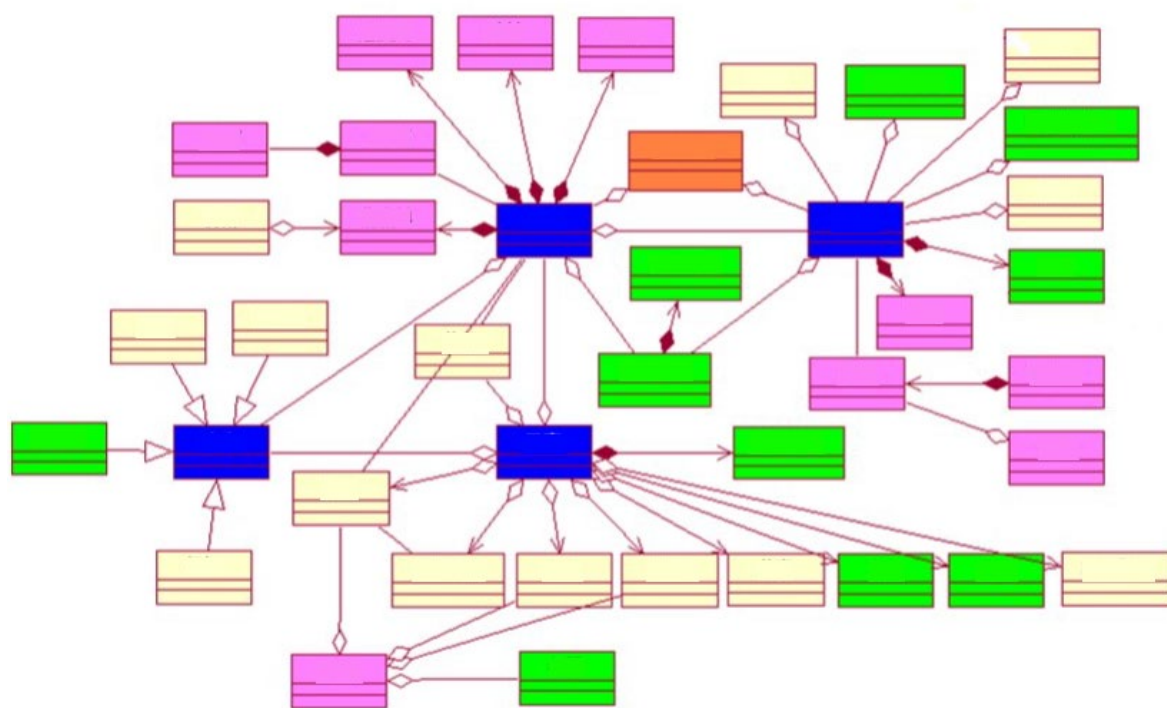


图 8-134 一张带有大量菱形的类图 (类的信息已隐去)

如果公司老总在充分调研员工能力以及公司业务的情况下, 着急过一把官瘾, 胡乱划部门, 提拔干部, 会大大损害所有人的利益, 很容易激起反抗。

然而, 如果 “架构师” 因为偷懒或炫耀, 胡乱定义组合 (聚合) 关联, **并不会激起各个代码片段的反抗**。计算机程序目前还没有产生自我

意识，没有 Neo（电影 “The Matrix”，《黑客帝国》），特别乖，爱怎么整都可以。

如图 8-135，如果建模为普通关联，还得给关联想个合适的名字。算了，懒得想，貌似说“订单有顾客”也说得通嘛，“有”那不就是组合（聚合）吗？干脆加个菱形吧，这样还省事，而且相对于一根直线，菱形让人有高大上的感觉！

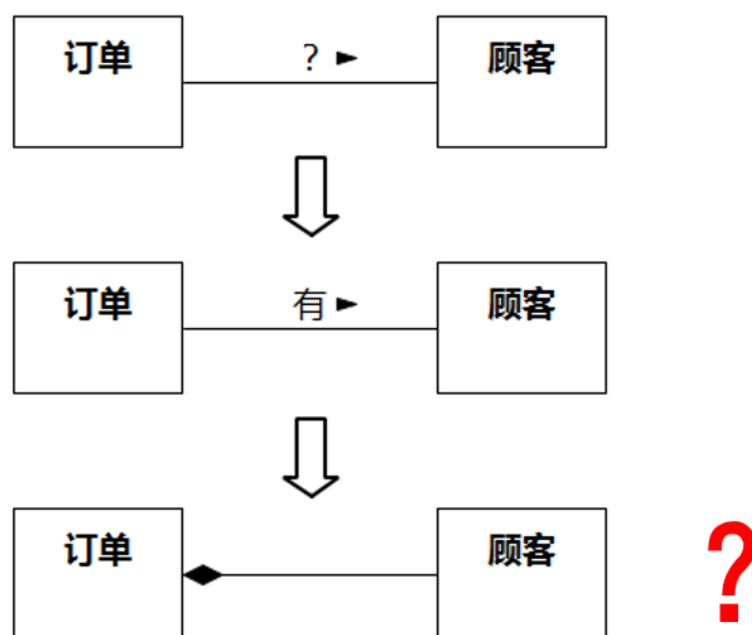


图 8-135 为了偷懒滥用组合（聚合）

最近一些年，由于 DDD 话语对“聚合”过度吹嘘，某些软件开发人员把“划分聚合”看成“有架构师能力”的表现，于是在没有足够证据的情况下，兴奋地把“聚合”到处用——哈哈，我会切割系统了，我架构师了！

这些人的思维经常是颠倒的：先拍脑袋定“聚合”，然后就按 DDD 话语的建议来使用，包括外部对象的访问、创建、访问数据等，然后再

用实现的代码 (show me the code 嘛) 来 “证明” 之前划分的 “聚合” 是正确的，形成 “完美” 闭环。

用公司类比，相当于公司老总拍脑袋把张三、李四、王五等人划分成一个部门，并任命张三为部门领导，然后通过张三发号施令，再用这个 “事实” 来 “证明” 张三作为部门领导是正确的。

当然，组合 (聚合) 和部门的类比不完全贴切，后文批评 “聚合根” 时还会提到。

建议一：有必要表达 “整体-部分” 关联时，仅使用组合，不使用聚合。

这一点和 Larman 是一致的。

我用下面的例子来说明本书的两条建议：

如图 8-136，把 “微信群” 和 “微信账户” 建模为聚合，而且多重性为多对多。理由是 “微信群有微信账户”，而且微信群解散，微信账户还在。

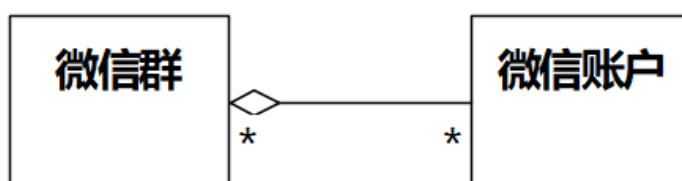


图 8-136 含糊的聚合

在没有足够证据时，应该建模为普通关联，如图 8-137：

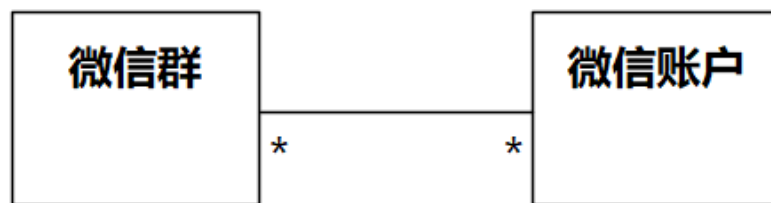


图 8-137 尽量使用普通关联

如果一定要使用“整体-部分”关联，使用如图 8-138 的组合。

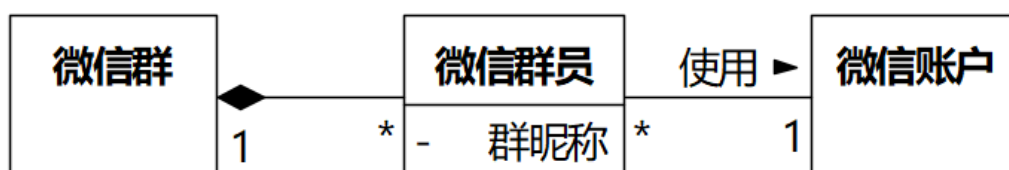


图 8-138 使用组合

图 8-138 将“微信群员”和“微信账户”分离，“微信群员”仅属于一个“微信群”。如果“微信群”对象消失，“微信群员”对象及相关属性值也就消失了，但“微信账户”还在。

注意，即使是图 8-138，也要有足够证据，而不是为了偷懒和炫耀。

如果只是玩文字游戏，图 8-138 也可以变成图 8-139，从另一个角度来“组合”似乎也未尝不可。

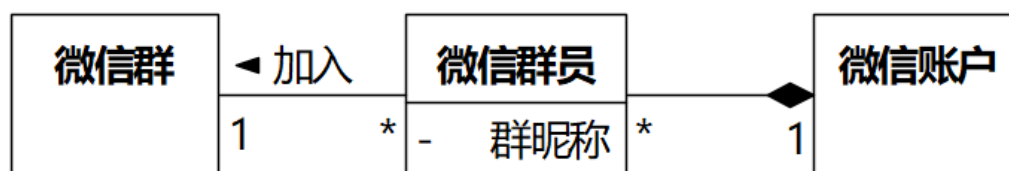


图 8-139 另一个角度的组合

8.3.3.4 类关系再整理

8.3.3.4.1 UML

有了前面的知识,我们需要再整理一下类的关系。用类图表示 UML 中类的关系如图 8-140。

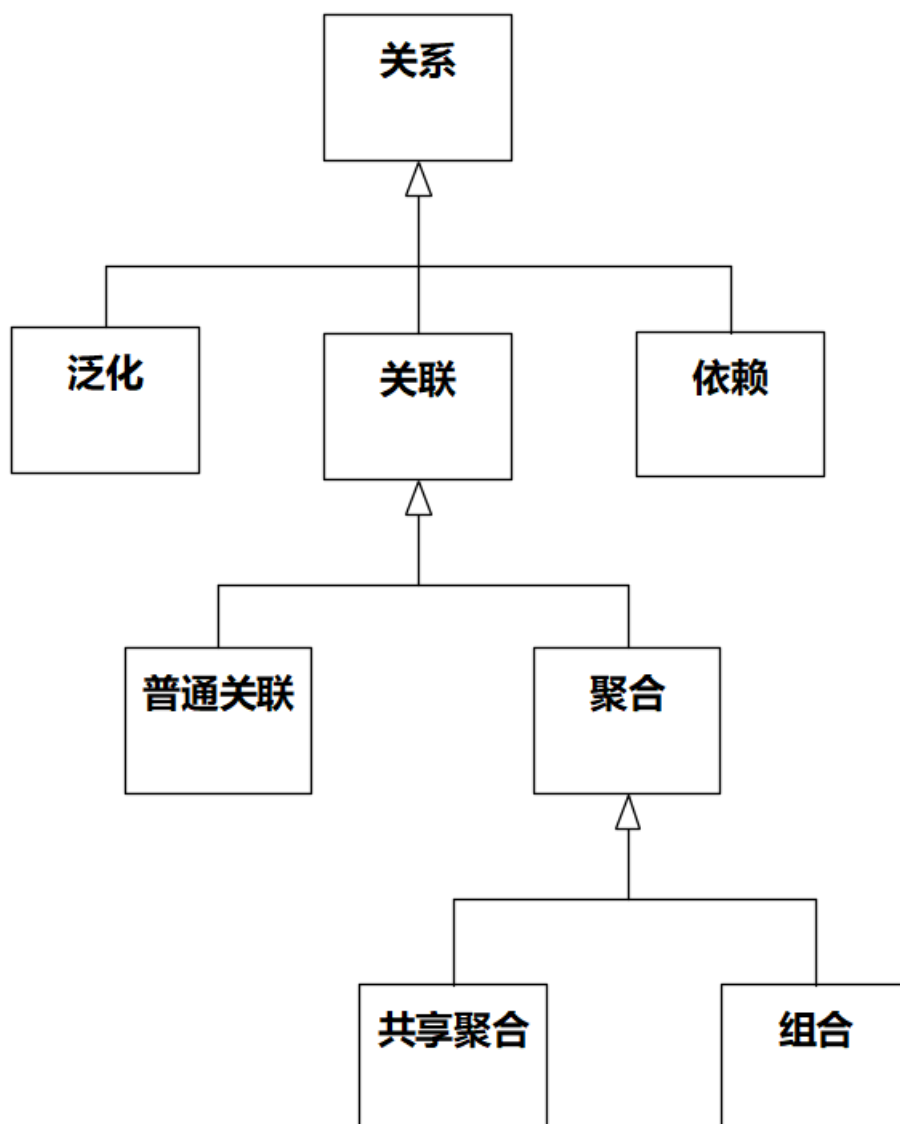


图 8-140 UML 中“类的关系”各概念之间的关系

从图 8-140 可以看到,泛化、关联和依赖在一个抽象级别,普通关联和聚合在一个抽象级别,共享聚合和组合(即独占聚合)在一个抽

象级别。因此，我们在表达的时候要注意，说“泛化和关联”可以，但说“泛化和聚合”、“泛化和组合”或“继承和组合”是不合适的。

8.3.3.4.2 本书的观点

本书的观点是取消“共享聚合”的概念，将其合并到“普通关联”中。此时，“组合”就等同于“聚合”，于是，“聚合”的概念也可以取消，得到图 8-141。

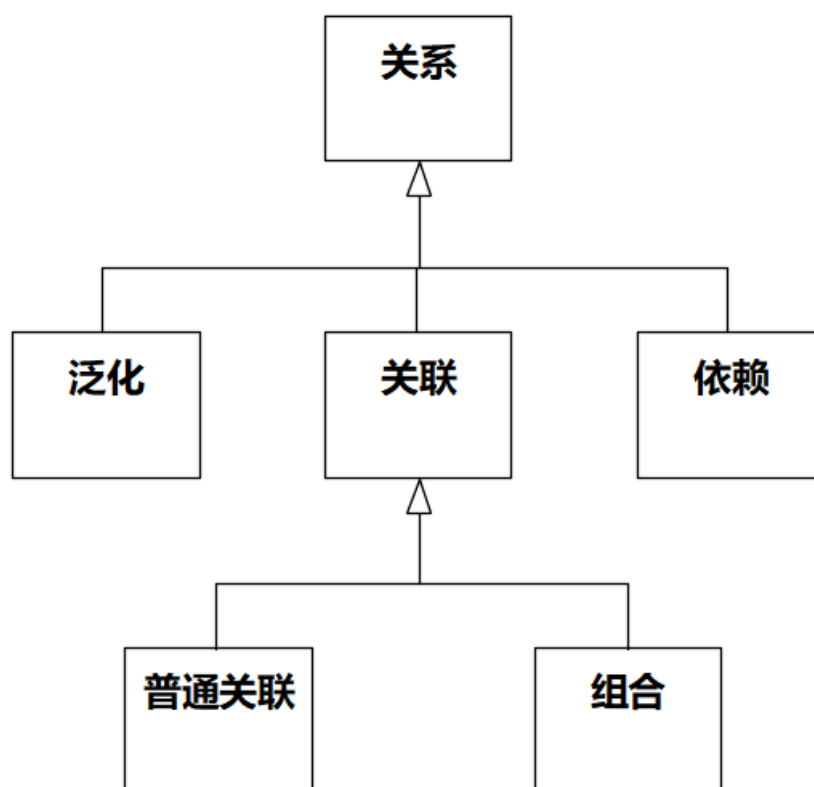


图 8-141 本书观点的类关系

8.3.3.4.3 和《设计模式》用语的区别

GoF 所写的“Design Patterns: Elements of Reusable Object-Oriented Software”（《设计模式》），第 1 章中有一句被广为流传的话：

Favor object composition over class inheritance.

优先使用对象组合而不是类继承。

这句话常让人误解组合和继承是一个级别的，其实，根据 GoF《设计模式》的用词，这句话中的“组合”应该近似于 UML 中的“关联”。

如图 8-142，在 GoF《设计模式》中，给出这句话之后，作者接下来讨论了 aggregation（聚合）和 acquaintance（认识）的区别，并且说 acquaintance 有时也被称为 association（关联）或 using（使用）。然而，在后面的内容中，作者把这几个词全部抛弃，一律使用 composition。

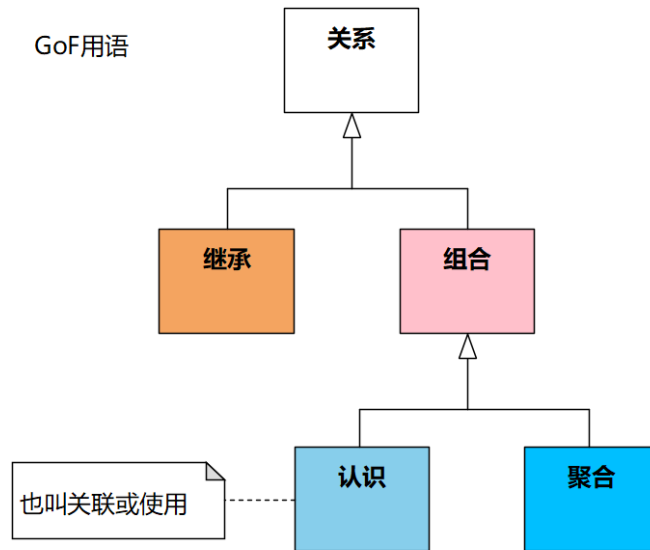
Consider the distinction between object [aggregation](#) and [acquaintance](#) and how differently they manifest themselves at compile- and run-times. Aggregation implies that one object owns or is responsible for another object. Generally we speak of an object *having* or being *part of* another object. Aggregation implies that an aggregate object and its owner have identical lifetimes.

Acquaintance implies that an object merely *knows of* another object. Sometimes acquaintance is called "[association](#)" or the "[using](#)" relationship. Acquainted objects may request operations of each other, but they aren't responsible for each other. Acquaintance is a weaker relationship than aggregation and suggests much looser coupling between objects.

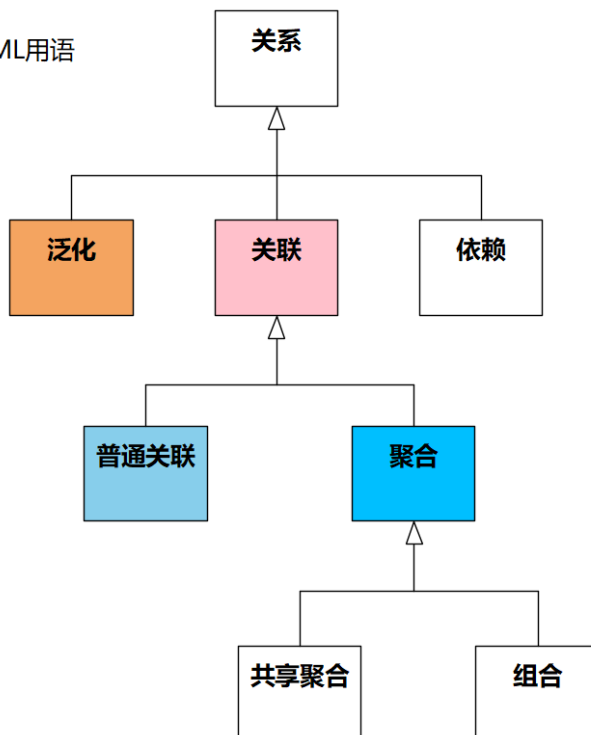
图 8-142 摘自 Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma , et al. , 1995

根据 GoF 书中内容猜测，其中用语和 UML 以及本书的用语的对应关系可能如图 8-143，图中我用相同颜色标出对应用语。

GoF用语



UML用语



本书用语

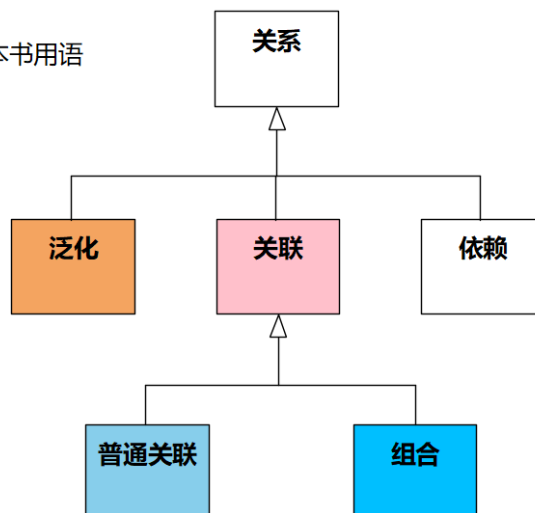


图 8-143 《设计模式》、UML 和本书用语的对应

以上仅属推测，而且书中的叙述也是有矛盾的，例如这一句：

Consider the distinction between object aggregation and acquaintance and how differently they manifest themselves at compile- and run-times.

考虑对象聚合和认识之间的区别，以及它们在编译时和运行时如何不同地展现自己。

这句话好像是在说“聚合”和“认识”在编译时和运行时有所不同，这和图 8-137 中 GoF 用语的“认识和聚合是组合的一种”矛盾。

另外，图 8-142 的片段中，把 association（关联）和 using（使用）说成同一个意思，这个也是让人困惑的。using 听起来更像是 UML 话语中的“依赖”，例如编程语言 C# 中的 using。

8.3.3.5 DDD 话语“聚合”中的伪创新

DDD 话语中也有“聚合”。Eric Evans 的“Domain-Driven Design: Tackling Complexity in the Heart of Software”书中是这样说的：

An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary.The root is the only member of the AGGREGATE that outside objects are allowed to hold references to,

一个 AGGREGATE 是一簇相关联的对象，我们把它作为数据变化

的单元来对待。每个 AGGREGATE 有一个根和一个边界。.....根是 AGGREGATE 的成员中唯一允许外部对象持有引用的,

以上文字提到了两个概念：

(1) aggregate (聚合体), 指整个聚合/组合结构。

(2) aggregate root (聚合根), 聚合/组合结构中扮演整体的对象。

先阐明本书对于这两个概念的观点：

两者都不提倡使用。其中, aggregate 属于冗余概念, aggregate root 属于错误概念。

接下来说一下原因。

我们以图 8-144 中 Grady Booch 关于系统结构的一段隐喻作为素材, 比较 aggregate、aggregate root 以及前文所说的 UML 的 aggregation (composition)。

例如, 如果为了跟踪主存储器中的定时问题, 就需要查看计算机的门一级的结构; 但是如果要求找出电子数据表格应用程序中的问题来源, 这个层次的抽象可能并不合适。

植物和动物的结构 在植物学中, 科学家们试图通过植物形态学的研究 (即研究它们的形态及结构) 来掌握植物之间的相似之处和不同之处。植物是复杂的多细胞有机体, 而各个植物器官系统的协同运作就产生了诸如光合作用和排泄作用这样的复杂行为。

植物包括 3 个主要结构——根、茎和叶, 每一个又有其自己的结构。例如, 根包括分根、根须、根尖及根帽。类似地, 从一片叶子的横截面也可以看到它的表皮、叶肉以及脉管组织。这些结构还可进一步再分成细胞, 而每个细胞内还有另一层次的复杂元素, 例如包含有叶绿素和细胞核等。与计算机结构相同, 植物的各部分也形成一个层次结构, 该结构中的每个层次又都有其自身的复杂性。

在同一层抽象上的所有组成部分都以周全的方式相互作用。例如, 在抽象的最高一层上,

图 8-144 摘自《面向对象分析与设计 (原书第 2 版)》, Grady Booch 著; 冯博琴 等 译, 英文原版出版于 1994 年

把图 8-144 的内容用类图和组合结构图表达如图 8-145，并标出三个用语的位置。

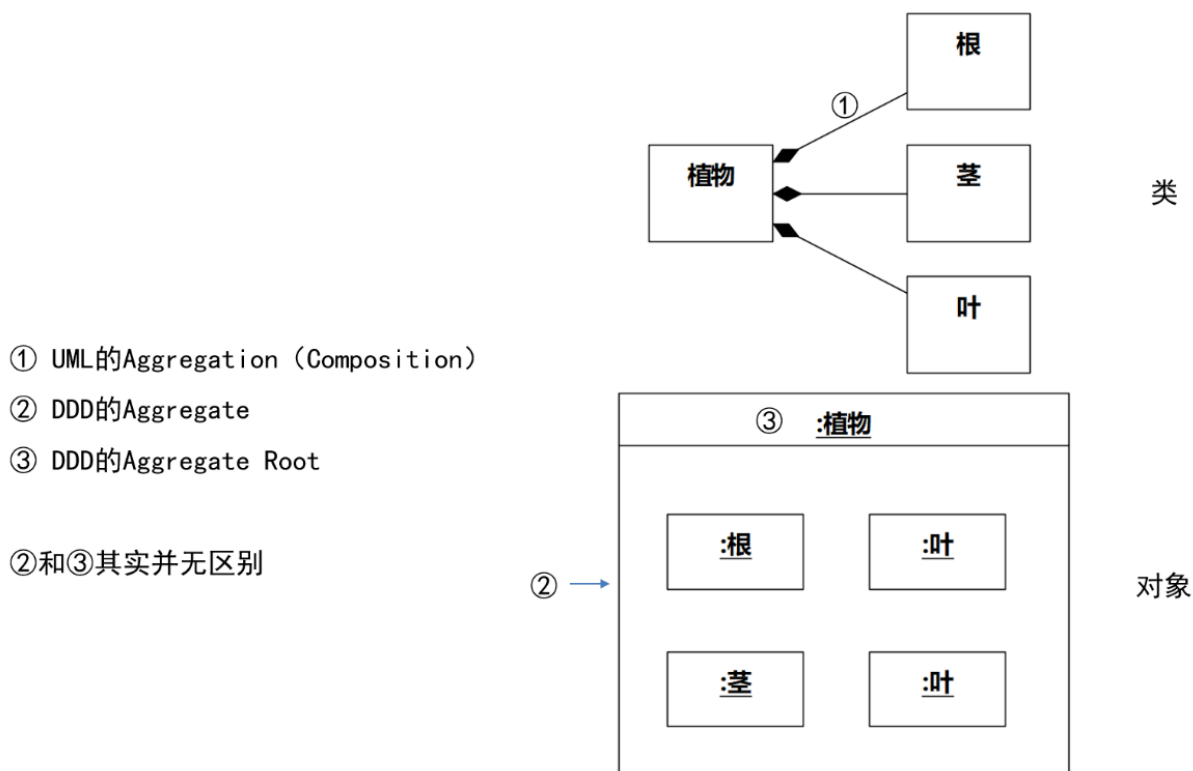


图 8-145 三个用语的位置

(1) aggregation 是本质概念

aggregation 指类之间的“整体-部分”关联，可以称为“聚合关联”。

例如，图 8-145 中，“植物”和“根”、“茎”、“叶”存在 aggregation 关联，说明可能会存在“植物”对象，它的组成部件是“根”、“茎”、“叶”对象。

根据前文提到的本书观点，“整体-部分”关联就是“组合 (composition)”，因此，图 8-145 以及后面的图形都会使用实心菱形。

(2) aggregate 是冗余概念

在面向对象建模领域，aggregate 并非 Eric Evans 在“Domain-Driven Design”书中首先使用。

1991 年，James Rumbaugh 等人的书“Object-Oriented Modeling and Design”中就阐述了 aggregation 和 aggregate，如图 8-146 所示。

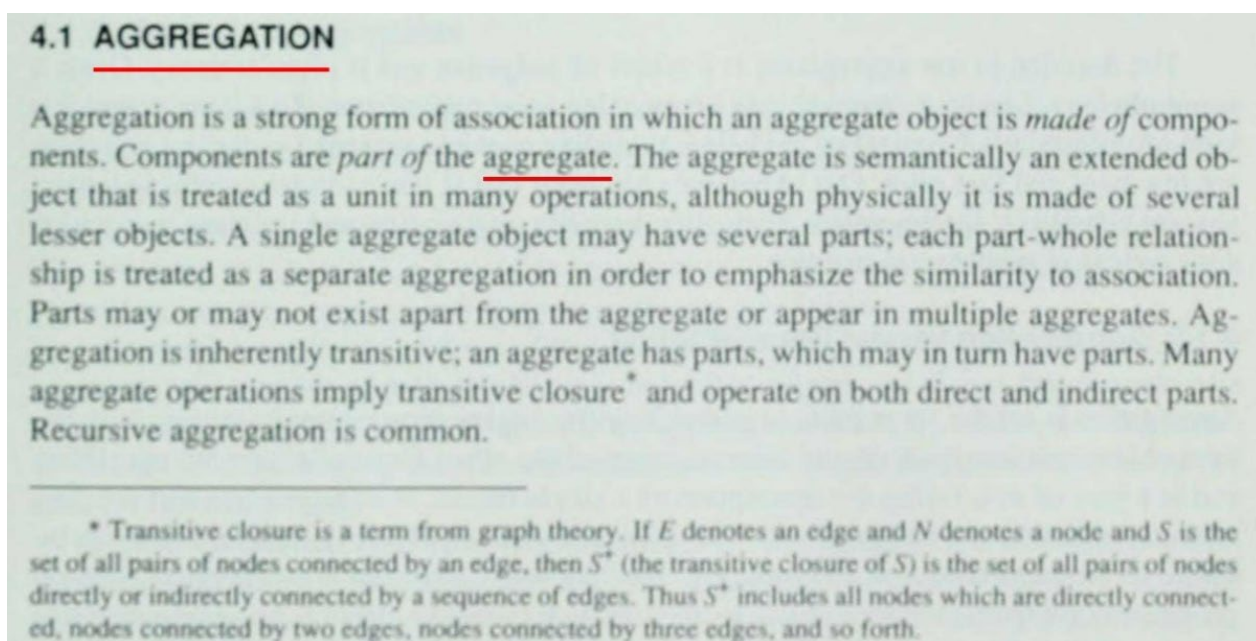


图 8-146 摘自 Object-Oriented Modeling and Design, James Rumbaugh et al. , 1991

可以看到,图 8-146 中的标题是 Aggregation,不是 Aggregate。

1999 年出版的“The Unified Modeling Language Reference Manual (UML 参考手册)”第 1 版中,也有“aggregate”词条,但内容只有一句话,接下来的“aggregation”词条的内容却达 4 页之多,如图 8-147。

aggregate

A class that represents the whole in an **aggregation** (whole-part) association.

aggregation

A form of **association** that specifies a whole-part relationship between an aggregate (a whole) and a constituent part.

See also composition.

Semantics

A **binary association** may be declared an aggregation—that is, a whole-part relationship. One end of the association is designated the aggregate while the other end is unmarked. Both ends may not be aggregates (or composites), but both ends can be unmarked (in which case, it is not an aggregation).

The links instantiated from aggregation associations obey certain rules. The aggregation relationship is transitive and antisymmetric across all aggregation links, even across those from different aggregation associations. Transitivity means that it makes sense to say that “B is part of A” if there is a path of aggregation links from B to A in the direction of traversal (in this example, from part to whole). Antisymmetry means that there are no cycles in the directed paths of aggregation links. That is, an object may not be directly or indirectly part of itself. Putting the two rules together, the graph of aggregation links from all aggregation associations forms a partial order graph, a graph without cycles (a tree is a special and common case of a partial order). **Figure 13-18** shows an example.

A directed path of links from object B to object A implies that there is a directed path of aggregation associations from the class of B to the class of A, but the path of associations may involve cycles in which the same class appears more than once. A directed path of aggregation associations from a class to itself is a *recursion*.

图 8-147 摘自 The Unified Modeling Language Reference Manual, James Rumbaugh et al. , 1999

可以看出来，UML 规范认为 aggregation 才是本质的概念。

为什么两者待遇不一样？

因为 aggregate 只是类 (类图上的结点) 在 aggregation 关联 (类图上的边) 中扮演的一个角色。同一个类可能在某个 aggregation 关联中扮演 aggregate (聚合体, 整体), 在另外一个 aggregation 关联中扮演 part (部件)。

我们进一步扩展图 8-145 中的类图，得到图 8-148。可以看到，包括“植物”在内的很多类，既扮演整体，也扮演部分。

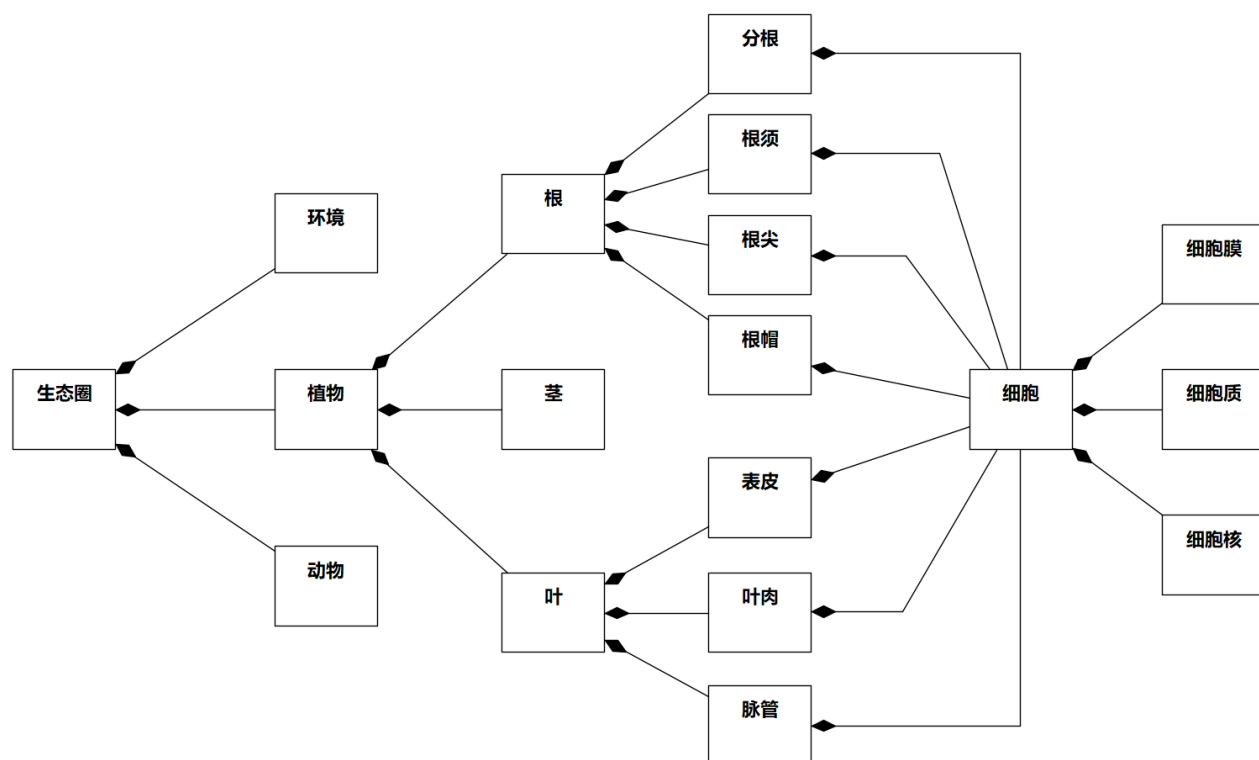


图 8-148 既扮演整体又扮演部分

一个对象可能由很多部件（最小部件是其简单属性值）组成，同时它又可以成为更大对象的部件。

离开特定的关联，指着一个对象说“它是整体”、“它是 aggregate”或“它是 aggregate root”，都是不合适的，除非只存在一级整体-部分结构。

这也是现在 DDD 实践中 aggregate 的现状——再多一级的话，不妨祭出“性能”遮羞布遮掩过去——“这得加载多少数据啊，会影响性能的！”。

即使只存在一级整体-部分结构，也没有必要在类上标注

“aggregate root”，或者圈一个边界说这是一个 aggregate，关系上的菱形标记已经提供了足够的信息。

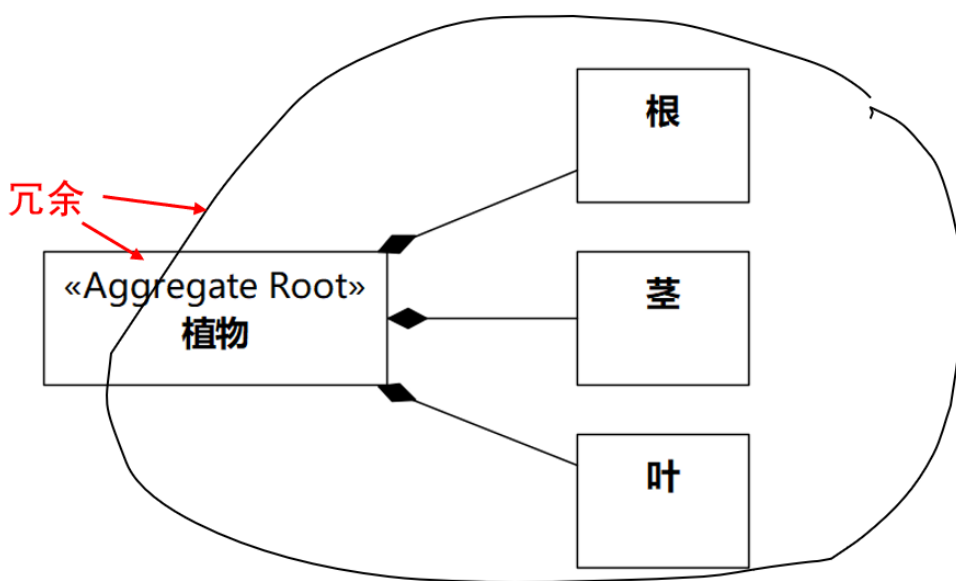


图 8-149 这两个标记都是冗余的

同理,如果要体现某些领域驱动设计实践投资少、见效快、产量高、门槛低、仪式感十足的特点,就可以考虑图 8-149 的标记。

(3) aggregate root 是伪创新

在 “Domain-Driven Design” 之前的软件开发书籍中, 没有出现 aggregate root 的说法, aggregate root 应该是 Eric Evans 的造词。

可惜, 这是一个伪创新。

我们再看一遍 “Domain-Driven Design” 中的描述:

An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each AGGREGATE has a root and a boundary.The root is the only member of the AGGREGATE that outside objects are allowed to

hold references to,

一个 AGGREGATE 是一簇相关联的对象，我们把它作为数据变化的单元来对待。每个 AGGREGATE 有一个根和一个边界。.....根是 AGGREGATE 的成员中唯一允许外部对象持有引用的，

以图 8-145 为例，按照这个说法，我们可以说：

①（“植物” + “根” + “茎” + “叶”）等一簇相关联的对象形成了一个 aggregate。

②其中“植物”是该 aggregate 的 aggregate root。

先来看①的问题。

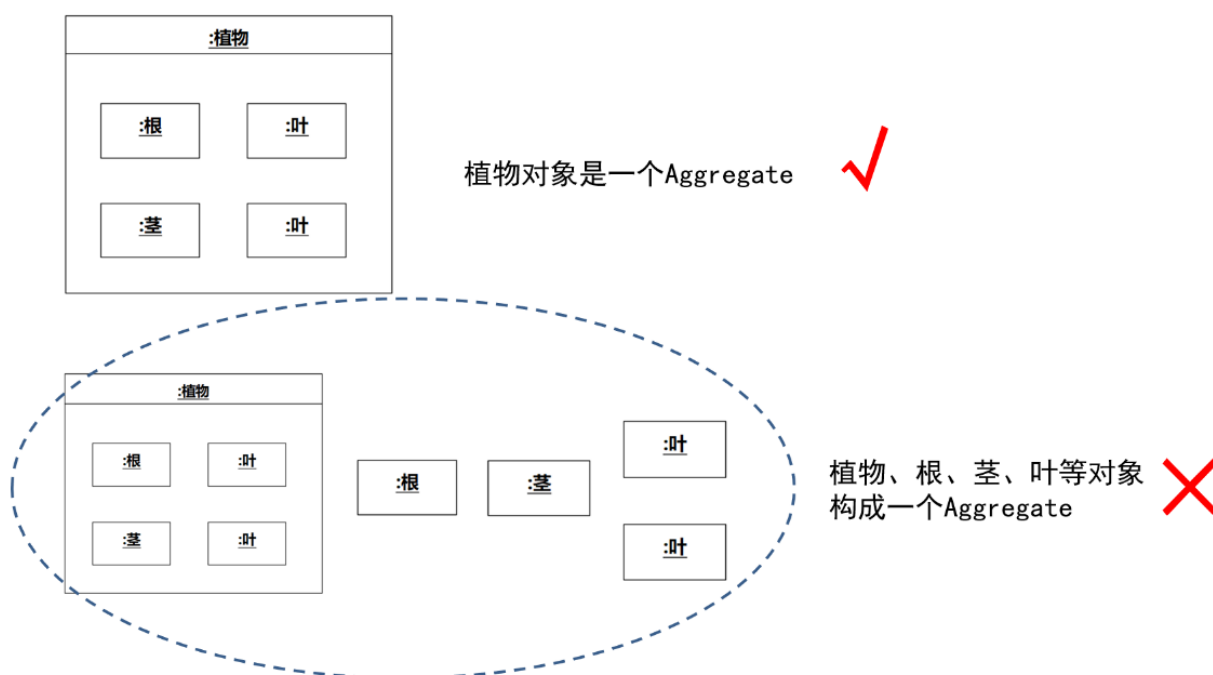


图 8-150 对象不需要也不能和其部件并列

如图 8-150，“植物”已经包括“根”、“茎”、“叶”等部件在内，不能也不需要再和这些部件并列。

在类级别，说“植物和根、茎、叶是整体-部分关联”，可以。

在对象级别，说“某个植物对象由若干根、茎、叶对象组成”，可以。

但是，说“某个植物对象和它的根、茎、叶一起组成 aggregate”，不可以。

说“植物对象是一个 aggregate”，可以，但这样的说法无意义。所有对象都是 aggregate，否则要它干什么呢？

要把这个圆过去，可以把“植物”排除在组成 aggregate 的“一簇相关联的对象”之外，说“一簇根、茎、叶对象组成了植物 aggregate”，不过，Eric Evans 又说了“根是 AGGREGATE 的成员”，看来是圆不过去了。

再来看②的问题。

即使我们排除了①的错误，采取“一簇根、茎、叶对象组成了植物 aggregate”的说法，但说法②仍然不合适。我们可以看下面的表述：

植物由根、茎、叶组成，所以，植物是[根.....]的根？

汽车由发动机、车身、底盘组成，所以，汽车是[发动机.....]的根？

墙由砖垒成，所以，墙是砖的根？

分子由原子组成，所以，分子是原子的根？

之所以会有 aggregate root 这样的错觉，有可能是受了关系数据库思考方式的影响。

图 8-145 的类图，如果用关系数据库来保存对象，可能会得到图 8-151 的几个表，这样就产生了“植物”就是“植物”表的错觉，于是觉得有必要提醒它是 aggregate root，别忘了带上其他几个小弟。

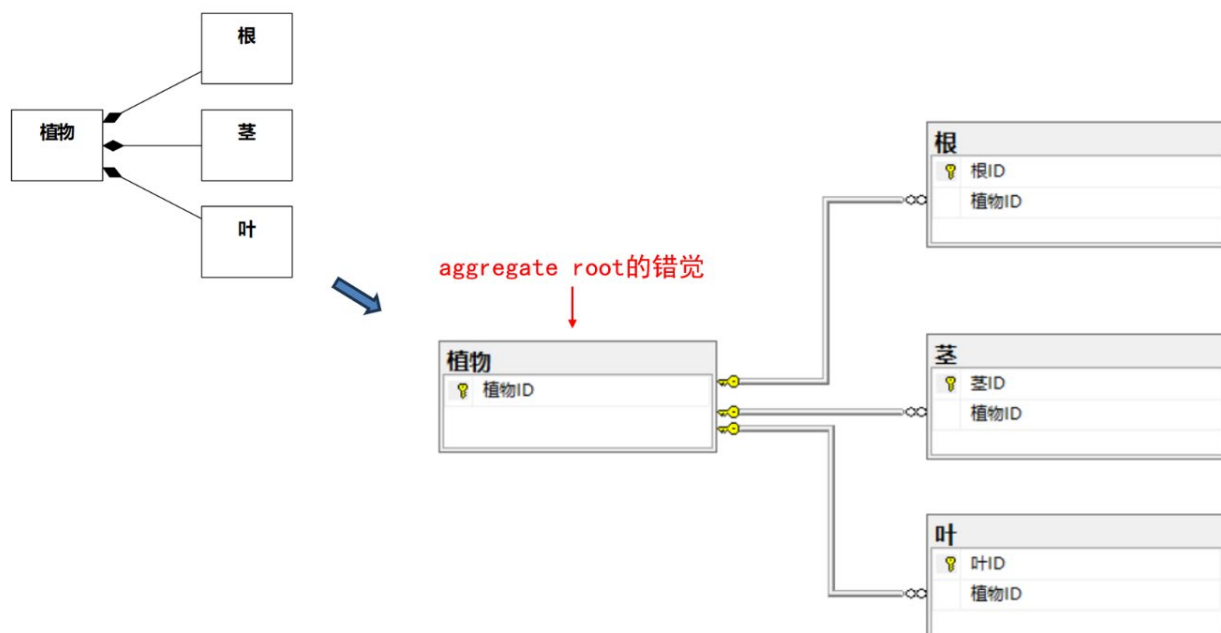


图 8-151 受关系数据库思考方式影响的 aggregate root 错觉

aggregate root 错觉另一个可能的原因来自人类社会的直觉。

在 8.3.3.3 关于“整体-部分”关联中，曾用部门结构来类比组合（聚合）：

给各部门分配大任务，部门把任务分解，再分配给部门内的各小组.....

这样的说法符合面向对象的思考，却与人类观察这个世界的直觉不符。

在直觉上，我们并不是观察到一个叫“项目部”的巨人，向着另一个巨人“财务部”吼一声，“喂，帮我结算这批临时工的工资！”，我们

观察到的是一个人“项目部经理”向另一个人“财务部经理”发出请求
“喂，帮我结算这批临时工的工资！”



图 8-152 人类直觉观察不到这样的现象

因为作为观察者的我们是人，或者说，是一个人脑系统，所以观察者直觉上得到的第一个抽象是和自己同一级别的抽象，即系统和系统的交互：项目部经理-财务部经理。

这样的场景也会诞生类似 aggregate root 的错觉，“项目部经理”是“项目部”的根，“财务部经理”是“财务部”的根，他们负责和其他部门打交道。

而在面向对象的世界中，对象是有“生命”和“智慧”的，“项目部”可以像人一样，向“财务部”发消息。

即使部门中确实有经理职位而且系统需要维护这个信息，在分配责任的时候，也并不需要把“经理”对象作为责任分配的起点。

★分不清现实中的“人”和系统中的“人”，这个现象在面向对象的初学者中很常见。例如，认为系统中“借书”是“馆员”或“读者”类的责任，因为现实中是由这两个“人”来完成的。曾经有领域驱动设计宏论：因为现实中事情是人做的，所以系统中映射“人”的类（例如“顾客”）优先作为“聚合根”云云。

此处还有一个有趣的问题，留给读者思考：

假设一定要推行“聚合根”这一领域驱动设计革命性创造，根据上文所说，我们直觉的级别，是系统和系统的交互：项目部经理<->财务部经理。

如果放大一个级别，可以得到组织和组织的交互：项目部<->财务部。

如果缩小一个级别，可以得到系统部件和系统部件的交互：项目部经理的嘴巴<->财务部经理的耳朵。

那么，“聚合根”项目部经理代表的是项目部呢？还是项目部经理的嘴巴呢？

(4) associated 的误用

我们第三次看“Domain-Driven Design”中的描述：

An AGGREGATE is a cluster of associated objects

一个 AGGREGATE 是一簇相关联的对象

前文讲到关联（Association）的概念时说过，关联是系统需要维护的关系。

我们用人来举例，如果是 “a cluster of associated objects（一簇相关联的对象）”，类图可能如图 8-153，各个部件之间存在关联：

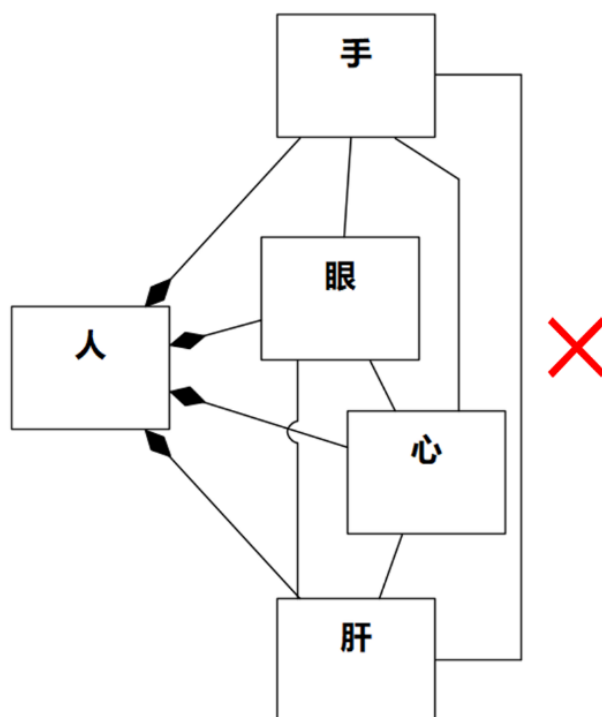


图 8-153 部件之间存在关联

虽然现实中的人体结构有可能是像图 8-153 这样的，但目前我们的软件结构更倾向于像图 8-154 这样，部件只和整体关联，部件之间（最好）不存在关联：

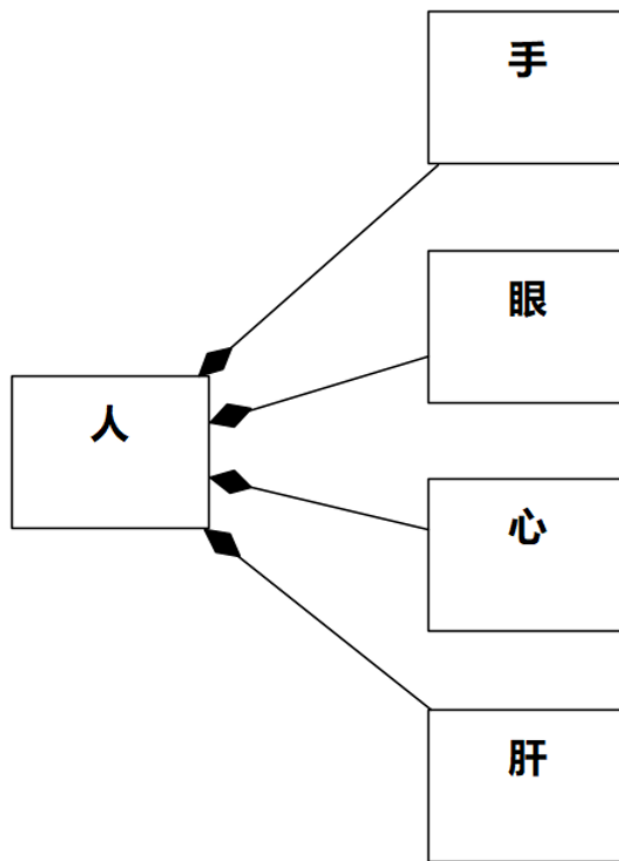


图 8-154 部件只和整体关联

可能有的读者会想，图 8-153 有道理呀，这些部件之间有“兄弟”或“同事”关联呀！这些“兄弟”或“同事”是可以从整体-部分关联推算出来的冗余概念，系统不需要维护。可参见“8.3.3.1 关联是不得不记住的关系”。

我们可以对比一下 Vaughn Vernon 在他的“Implementing Domain-Driven Design” [Vernon 2013]中的用词：

Is an Aggregate just a way to cluster a graph of closely related objects under a common parent? If so, is there some practical limit to the number of objects that should be allowed to reside in the graph? Since one Aggregate instance can

*reference other Aggregate instances, can the **associations** be navigated deeply, modifying various objects along the way?*

*聚合只是把在共同父对象之下紧密**相关**的对象图聚集在一起的一种方式吗？如果是这样，对于允许驻留在该图上的对象数目，有没有一些实际的限制？既然一个聚合实例可以引用其他聚合实例，那么**关联**是否可以向深处导航，沿途修改各种对象呢？*

★ “Implementing Domain-Driven Design” 的中译本《实现领域驱动设计》在此处的翻译存在严重错误，参见 《实现领域驱动设计》的译者其实没错？（二）

Vaughn Vernon 在此处用词非常准确。

描述 “一簇对象” 时，他用的词是 “related（相关的）”。

怎么个相关？静态上，它们都是同一个整体的部件，动态上，它们中的部分或全部会在某个场景中由整体对象协调来完成任务，如图 8-155：

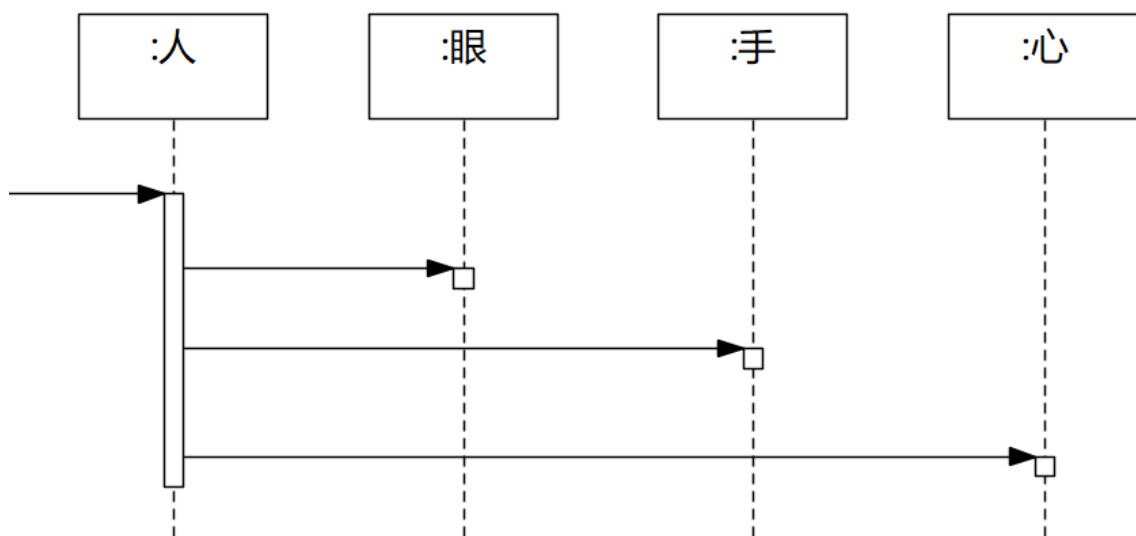


图 8-155 部件在整体的协调下协作完成任务

同一段文字中，Vernon 也在正确的地方使用了 “association (关联)” 一词，讨论沿着关联关系导航的深度问题。

读者可以回看 “8.3.1.2 区分几个术语” 中与 “关系 (relationship)” 和 “关联 (association)” 相关的内容，以对比 related 和 associated。

(5) Eric Evans 的葡萄隐喻错误

软件开发中，比 UML、DDD 更早使用 Aggregate 这个词的是 SQL，各种 Aggregate 函数如 SUM()、AVG()、MAX()、MIN()，用于统计表中的数据。Aggregate 这个用词经常会让人产生错觉，以为是 “累计” 或 “集合”。

Eric Evans 在 “Domain-Driven Design” 书中用一串葡萄来隐喻 Aggregate，如图 8-156。这个隐喻的选择是有问题的，有可能把 “聚合” 当成了 “集合”。

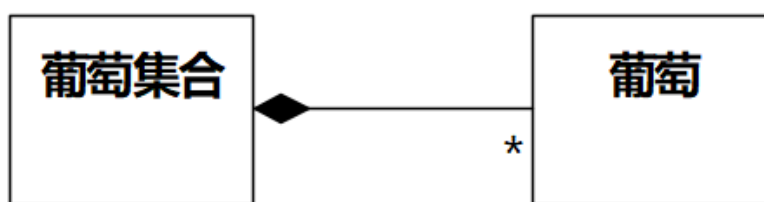
Aggregates



Minimalist design of associations helps simplify traversal and limit the

图8-156 摘自 Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans , 2003

一串葡萄就算有一亿颗，也只是同一个类“葡萄”的对象集合。在“8.2.4.7 类命名用单数”中说过，不需要对纯粹的对象集合建模。



8-157 不需要建模纯粹的“葡萄集合”

★当然，此处适合践行某些领域驱动设计实践投资少、见效快、产量高、门槛低、仪式感十足的精神，可以把每个类都无条件地配一个“**们”或“**s”，类的个数瞬间翻倍。

如果一定要以葡萄做素材，以下这个隐喻可能更合理：

若干颗葡萄、若干个煎蛋、若干根油条.....组成一份早餐，其中维生素 C 的质量不得少于蛋白质质量的 1%.....这个更有意义，如图 8-158。

(注：该图仅从图 8-156 的葡萄延伸以作说明，图中的领域知识未经过调研，“葡萄”、“煎蛋”、“0.01”等抽象程度也不够，烦请忽略这些问题。)

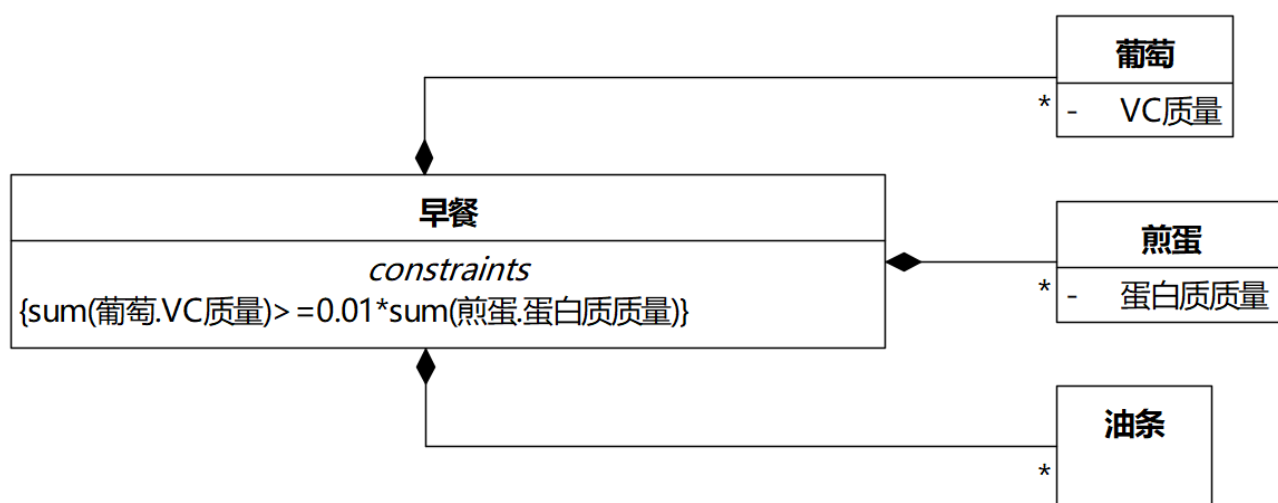


图 8-158 更合理的隐喻

如果“对象集合”另有含义，演化成了另一个类的对象，那么这个“另一个类”不会仅仅是一个纯粹的集合，应该还会封装其他内容。

常被用于举例的“订单”和“订单项”，平时我们看到的例子可能如图 8-159：

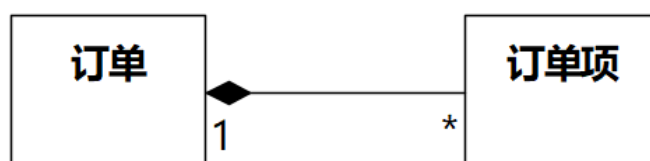


图 8-159 常见的订单-订单项类图

但这只是简化版，如果仅是这样，“订单”就没有必要存在了。之所以需要“订单”的概念，是因为还有“下单日期”、“顾客”、“收货地址”等概念，要通过“订单”组织起来，如图 8-160。

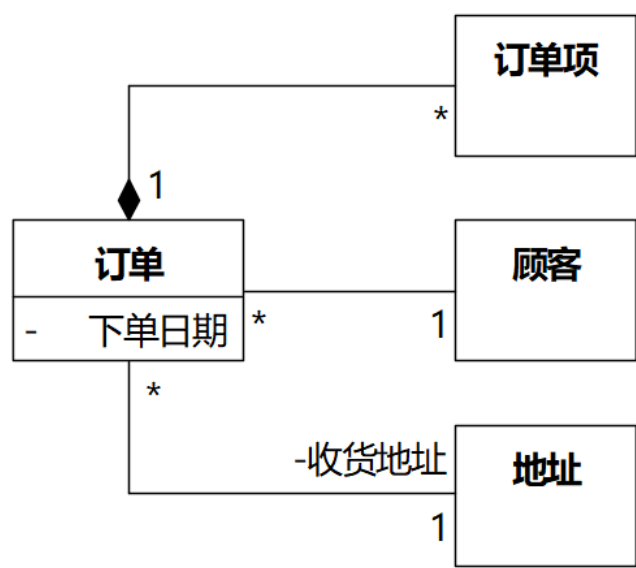


图 8-160 扮演整体的类还会封装其他知识

Eric Evans 选用这个葡萄图片，可能还搞错了另一个知识。这个知识不是软件开发知识，而是植物学知识。

植物学上有聚合果（Aggregate Fruit）的概念，如图 8-161：

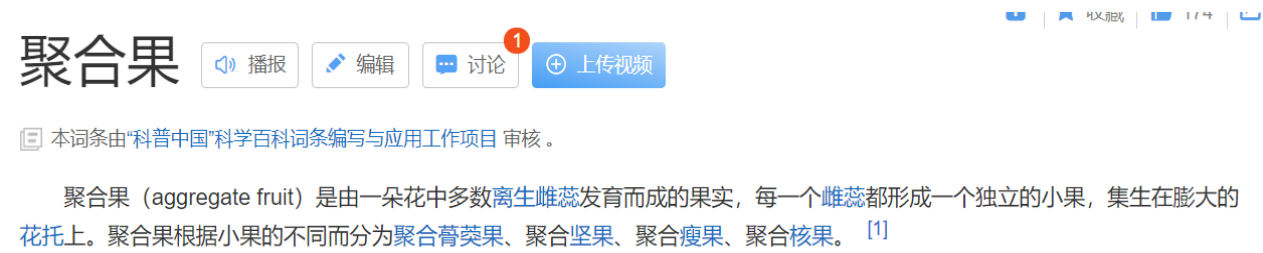


图 8-161 摘自百度百科“聚合果”词条

Eric Evans 可能想到“Aggregate Fruit”这个术语，觉得葡萄是成串的，以为葡萄是“Aggregate Fruit”，于是把图片放上去了。

其实葡萄是单果，如图 8-162。

单果的代表是葡萄、西瓜、橙子以及椰子，这些果实无不例外都是单个就能成果的状态。单果由单朵受粉花产生的子房形成，即使是成串生产的葡萄也属于单果，这是因为每粒葡萄都是单一花朵子房的产物。而聚合果不同的是，聚合果由单一受粉花产生的多个子房形成，由许多微球体构成的树莓是典型的聚合果，这种水果也被叫做附果，其它还有草莓、向日葵等都属于这个范畴。



图 8-162 摘自 <https://zhuanlan.zhihu.com/p/37538771>

8.3.4 识别泛化关系

8.3.4.1 识别泛化的思路

8.3.4.1.1 直接形成

类图中的两个类可能会直接形成泛化关系，如图 8-163 所示。和识别关联相似，我们可以针对每两个类，思考“A 是 B 的一种吗？” ，再反过来思考“B 是 A 的一种吗？” 如果类图中有 n 个类，就需要思考 $2C_n^2 = n(n-1)$ 次。n=11 时，就是 110 次。

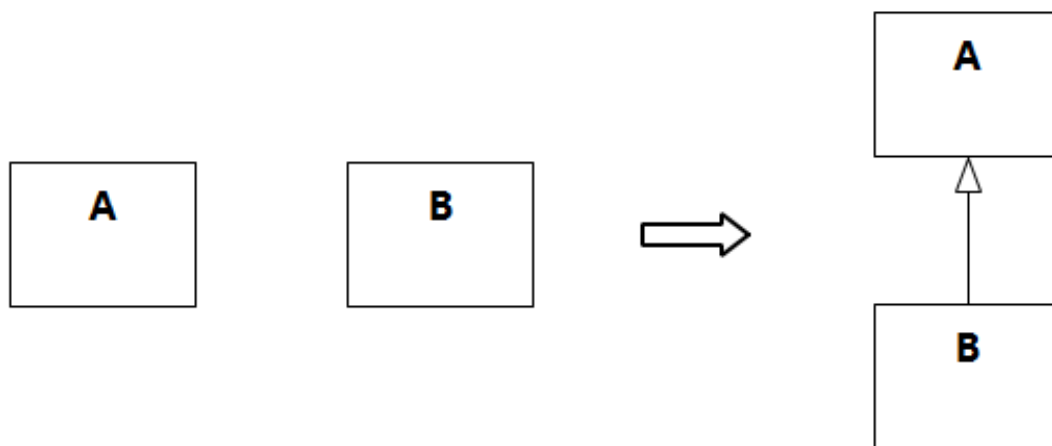


图 8-163 直接形成-两个类之间直接形成泛化关系

实际上，类图上已有的两个类有泛化关系但未识别的情况并不多，因为之前在识别类和属性以及识别类之间的关联的时候很有可能已经发现了。

8.3.4.1.1 自下而上（从特殊到一般）

更多的情况是发现类图上已有的两个或多个类有共同特征，于是抽象出共同的超类，如图 8-164 所示。

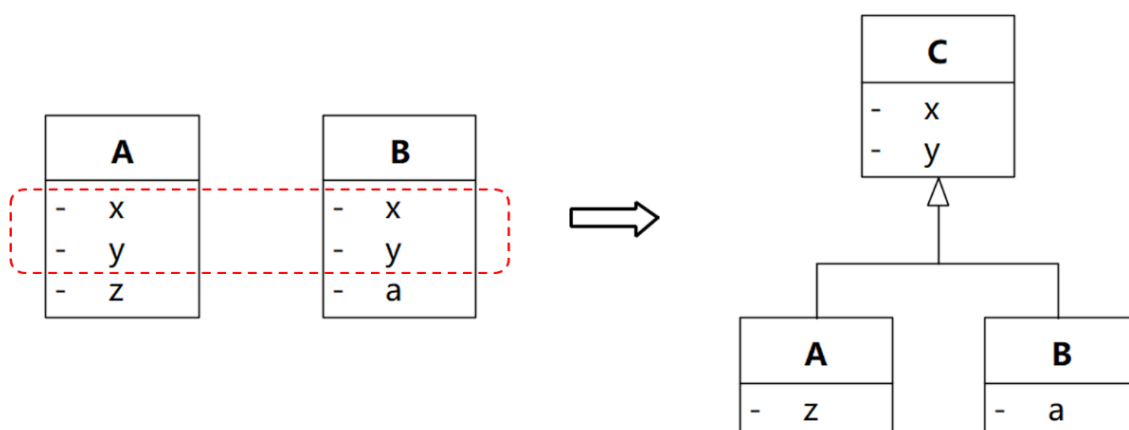


图 8-164 自下而上-两个类之上有共同的超类

关联也可以看作类的属性，关联的角色名相当于类的属性名称。如

果多个类关联到同一个类而且角色名相同,也可以考虑泛化出共同的超类,如图 8-165。注意,角色名要相同,否则即使类型相同也不是同一属性。

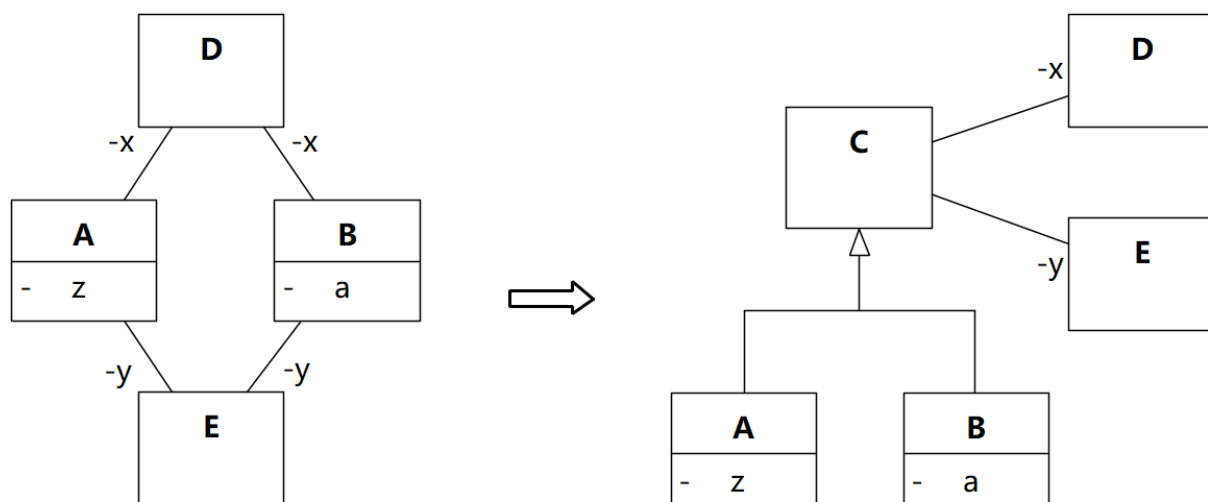


图 8-165 共同的关联也可以提炼超类

8.3.4.1.3 自上而下 (从一般到特殊)

如图 8-166 所示,这个识别思路就是 8.2.5.6 属性是否对所有对象都有意义里的思路。

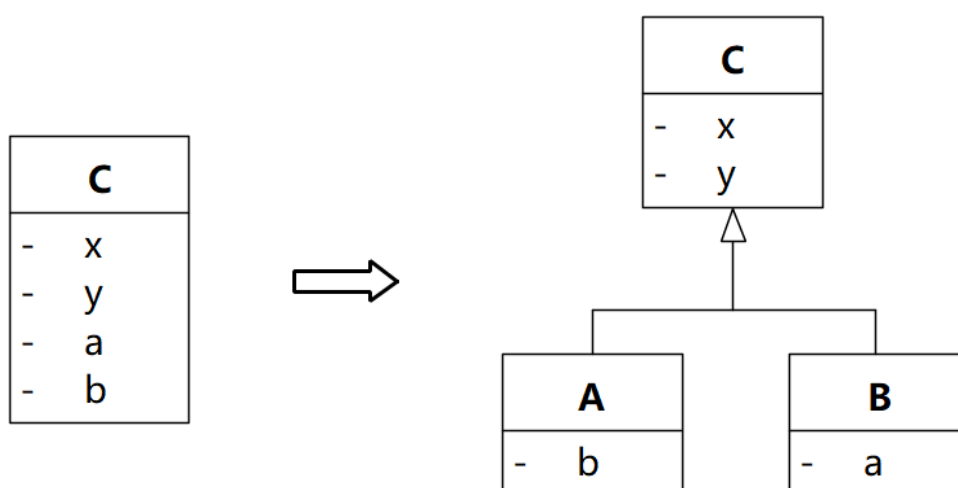


图 8-166 自上而下-一个类分裂出子类

和上面的“自下而上”一样, 这里的属性也包括关联, 如图 8-167。

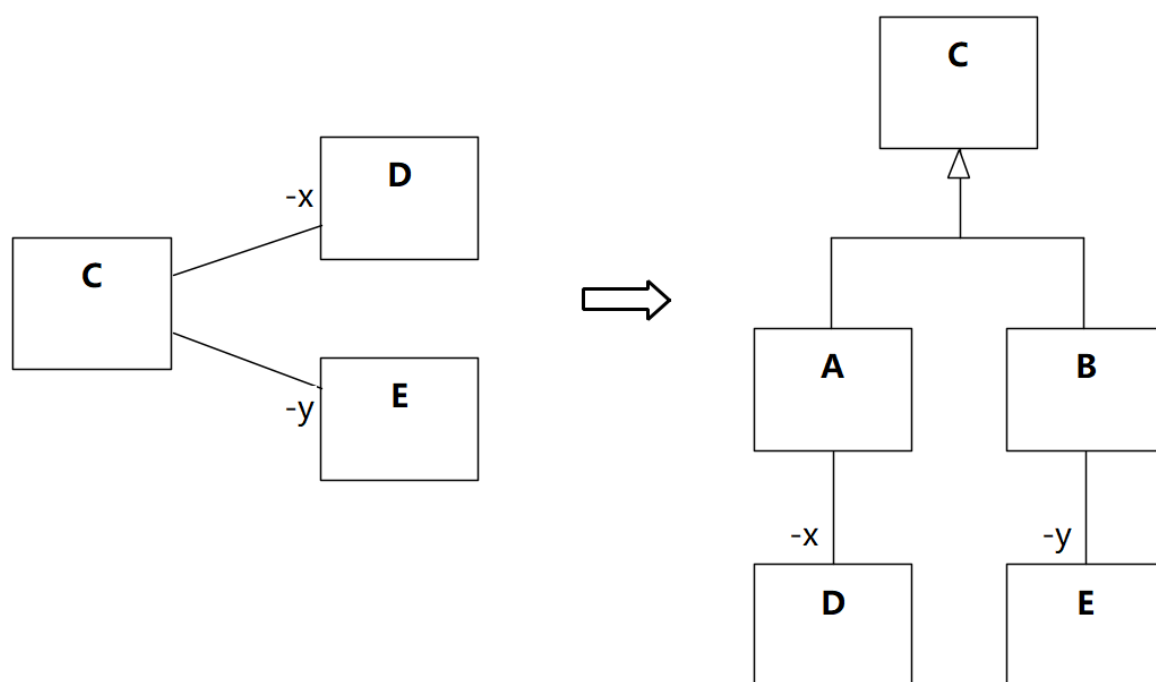


图 8-167 关联只对部分对象有意义时分出子类

不一定像图 8-167 那样关联到不同的类。关联到同一个类但角色名不同, 而某个角色只对部分对象有意义, 也可以考虑分出子类, 如图 8-168。

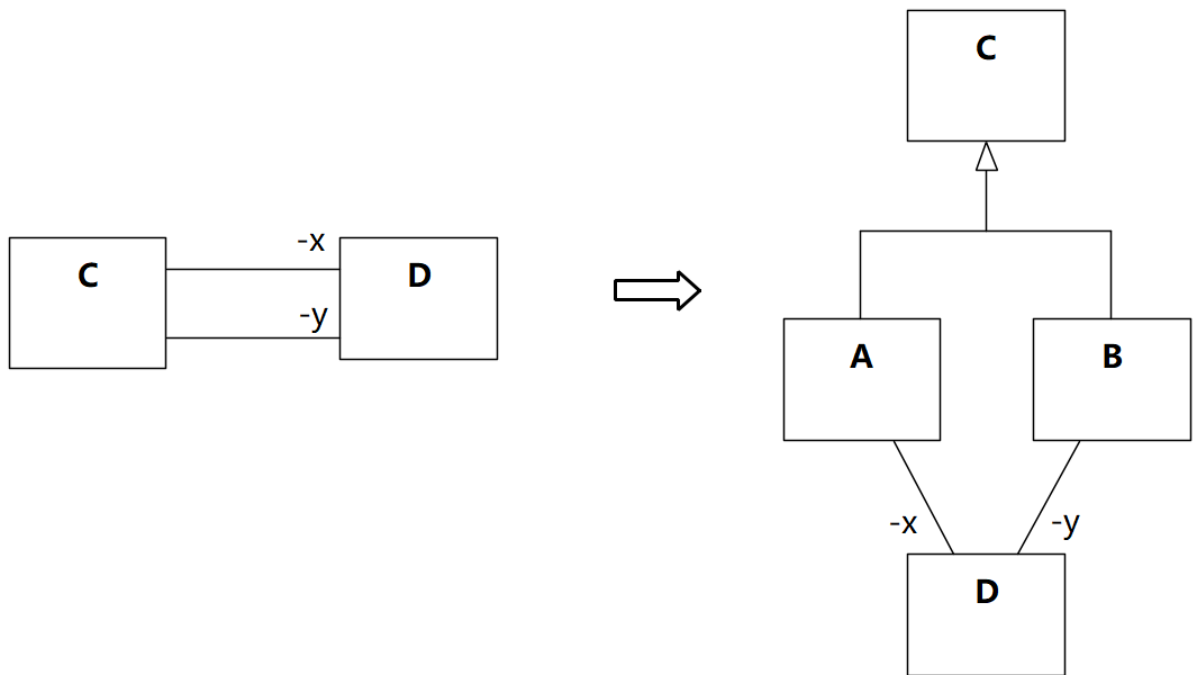


图 8-168 某个角色只对部分对象有意义时分出子类

8.3.4.2 警惕拼凑泛化

您可能注意到，以上我们尽量通过属性（包括关联）来解释泛化关系。虽然泛化带来的好处是落在行为上，但如果抛开属性直扑行为，很可能会带来“伪泛化”、“伪面向对象”。

如图 8-169，因为 X 和 Y 都有操作 op1，所以泛化出 A，把 op1 提上去成为抽象操作。这个没有问题。

问题出在前面，怎么知道 op1 作为 X 和 Y 的操作是合适的？最终的依据还是 X 和 Y 的属性（包括关联）。

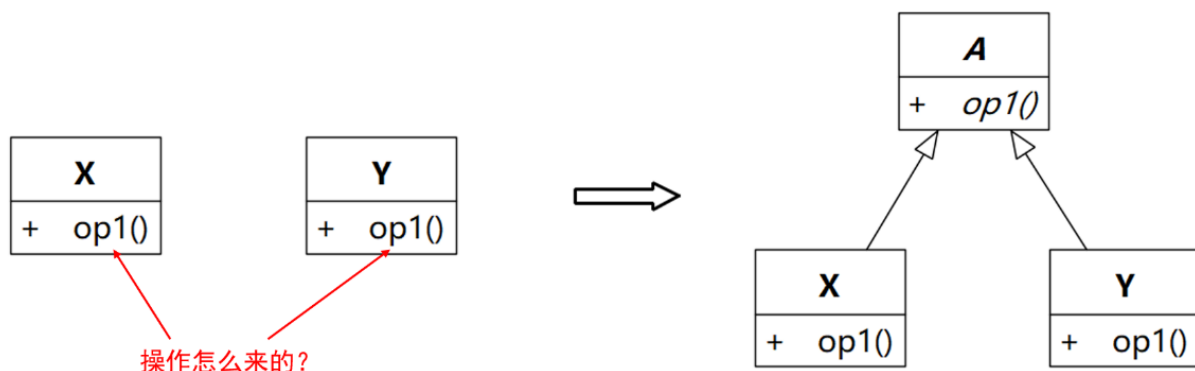


图 8-169 操作怎么来的，需要有依据

有一种偷懒遮羞布，就是胡乱安排操作，然后拼凑出泛化关系，根本不顾安排的操作是否合理。

一些伪面向对象实践经常得到这样的类：它的名称的最后是 or 或 er（汉语则为“器”）。

开发人员可能一开始按照面向过程的思路噼里啪啦把代码写出来，然后出于赶时髦需要“面向对象”，他就把过程名称加上 or 或 er 作为类名称，然后把原来的过程作为 or 或 er 类的操作。当存在多个类似过程时，加上一些泛化关系来做点缀，这样看起来就更有“面向对象”的味道了。

更赤裸裸的，连类都不要了，直接把行为变成“**接口”、“**able”或“可**”。典型的例子是常被用于 GoF 模式举例的 Flyable(可飞行)。然后往“**接口”、“**able”或“可**”上面拼凑“实现”关系，相当于泛化关系的变体。

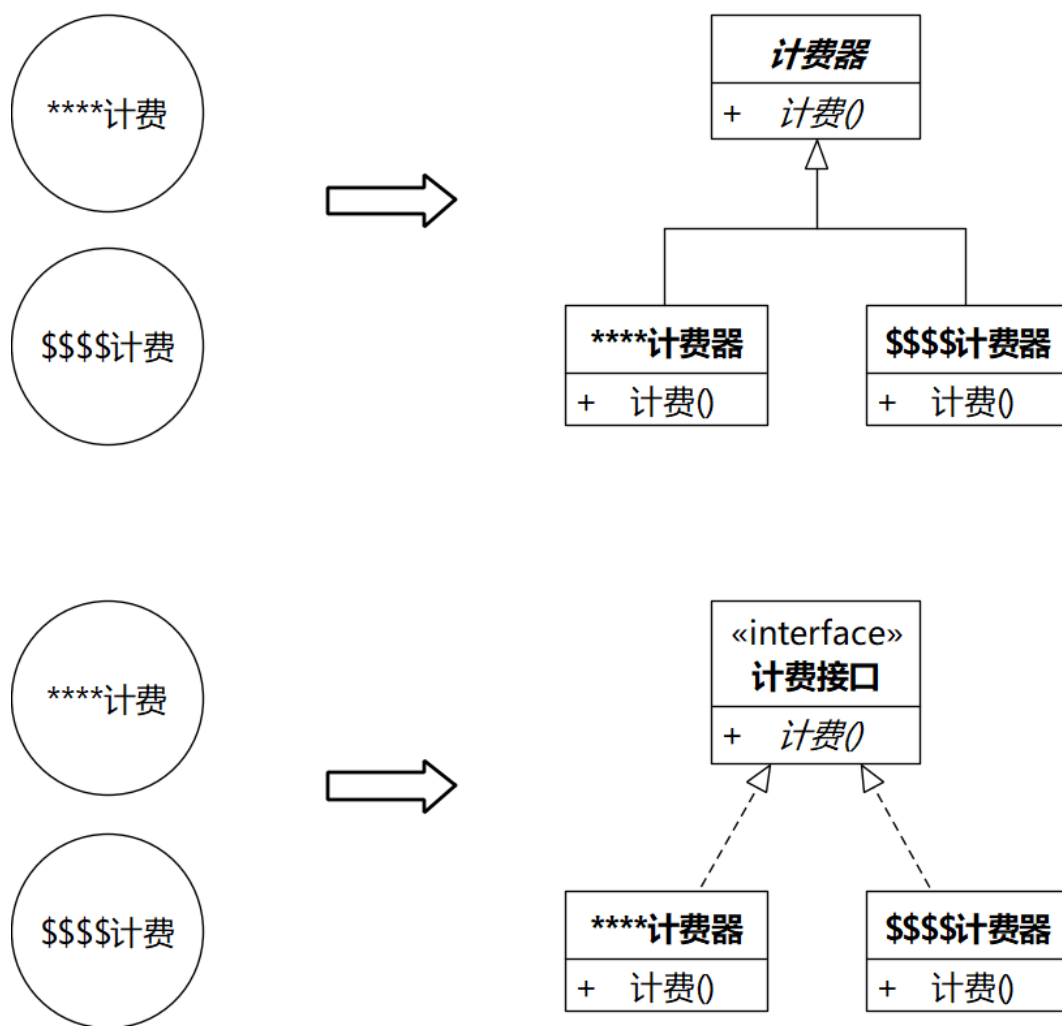


图 8-170 带有泛化关系的 or 或 er 类

可能在某些开发人员眼里，图 8-170 很有格调，可以用来吹嘘的高大上词汇有：OCP（开放-关闭原则）、DIP（依赖倒置原则）、模板方法模式.....等。这些由泛化关系衍生出来，被网红圈子广泛吹嘘的原则和模式作用十分有限，指望了解被包装出来的“SOLID 原则”之类就能应对软件复杂性，那真是太天真了。

★关于 Robert C. Martin 和他的书《敏捷软件开发-原则、方法与实践》的评价，参见第 1 章关于“敏捷”染色的阐述。

or 或 er 类往往没有属性，只有操作，大量的逻辑仍然隐藏在子类

操作的实现中。当然，开发人员也可能觉得这是好事，“这说明我有算法啊！”，好像很高大上，但这所谓的“算法”（其实是核心域逻辑）正是建模的重点，结果被开发人员完美地遮掩过去了。

和其他的偷懒遮羞布类似，这种 or 或 er 的做法——对应，思考工作量小，还有各种高大上词汇护法，于是开发人员洋洋得意，感觉自己已经很厉害了，连称“受用”，纷纷去拥抱这种偷懒遮羞布。

or 或 er 类有时会使用“策略模式”作为伪装，如图 8-171，哇，我可以灵活组装各种策略！顺便再吹一通“组合优于继承”之类，其实还是换汤不换药，核心域逻辑仍然隐藏在子类操作的实现中。那些“策略”才是建模的重点，同样被开发人员完美地遮掩过去了。

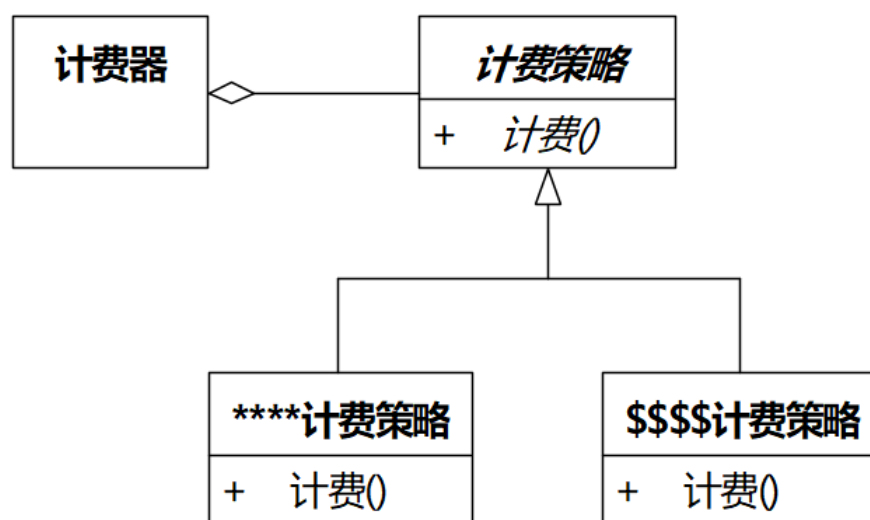


图 8-171 “策略模式”换汤不换药

回到前面所说的，泛化是集合关系。如果泛化结构中的类是类似上面的这些类，它们封装行为，却没有属性，“类的对象集合”没有了意义。泛化关系的产生已经脱离核心域知识，很容易出现第 1 章所说的“思维颠倒”的现象。

如果一个类的命名中有类似这些内容:er、or、器、策略、Strategy、Policy、规则、Rule、算法、Algorithm.....然后这个类或其子类的操作中有长长的“算法”,那么应该思考一下,长长的“算法”中到底定义了哪些变量?哪个部分的代码最复杂?

背后往往就是候选的实体类以及需要封装的操作,尽量分离出实体类,把各种逻辑尽量封装在实体类的操作中。**这样的思考更辛苦,但也更有价值。**

有心的读者还可以仔细观察一下你接触到的领域驱动设计相关的文章,看看有没有这样的现象:

全篇文章没有剖析复杂一点的逻辑,但会有“**策略”、“**规则”这样的类或者组件存在。莫非这些“**策略”、“**规则”是别人已经做好的,他拿来就用? **追问下去,多半不是,而是他要负责解决的问题。**

你猜我怎么知道是这样?因为我接触的开发团队和开发人员太多了,追问下去,脓包破裂的概率极高。

8.3.5 泛化的一些重点讨论

8.3.5.1 子集的不相交和完整

泛化是集合关系,在建模泛化关系时,我们对泛化关系中的子类(子集)的要求应该严格到什么程度?例如,不同子集可以有相同的元素吗?子集的并集必须等于全集吗?

UML 规范对此持宽松的态度,泛化关系缺省是无约束的。如果要表达一个元素只能出现在一个子集中,需要给泛化关系加{disjoint}约

束，如果要表达一个元素必须出现在一个子集中，需要给泛化关系加 {complete}约束，如图 8-172。

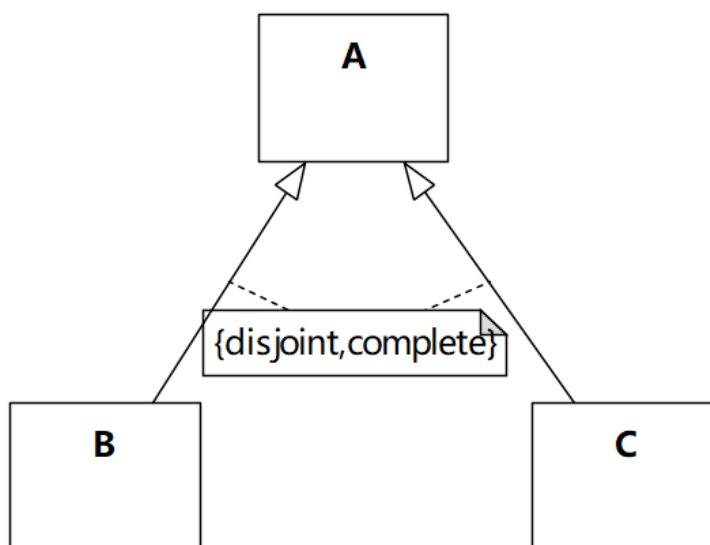


图 8-172 泛化的不相交和完整约束

(1) 不相交

本书的观点是：同一个泛化集中的任意子类之间不相交，应该是缺省的做法。或者说，尽量把泛化建立在不相交的类之上。

例如，自然语言“员工有调度员、装卸工、配货员”，可能有的人画成图 8-173 的组合关联：

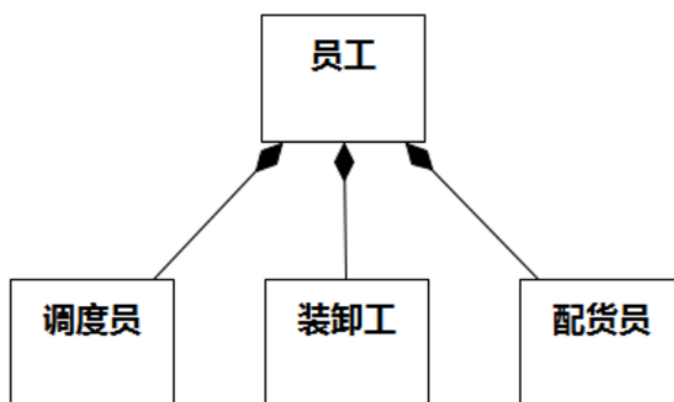


图 8-173 错误的关联关系

这当然是错误的。“员工有调度员、装卸工、配货员”指的是“员工”的对象集合包含了“调度员”、“装卸工”、“配货员”的对象集合，不是指一个“员工”对象由“调度员”对象、“装卸工”对象、“配货员”对象组成。这个“有”是集合的“有”，是泛化关系，如图 8-174。

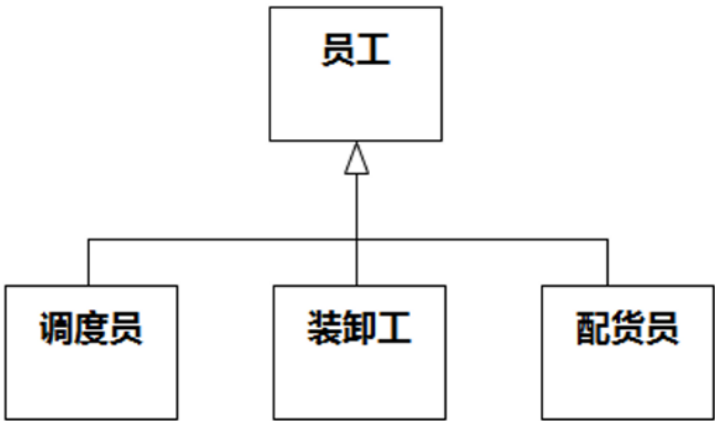


图 8-174 泛化关系

但图 8-174 也不是合适的建模结果，因为很可能允许员工身兼数职，同时既是调度员也是配货员。此时，“员工”的任意子集之间的交集不再为空，应该调整泛化结构所在的位置，例如改为图 8-175。

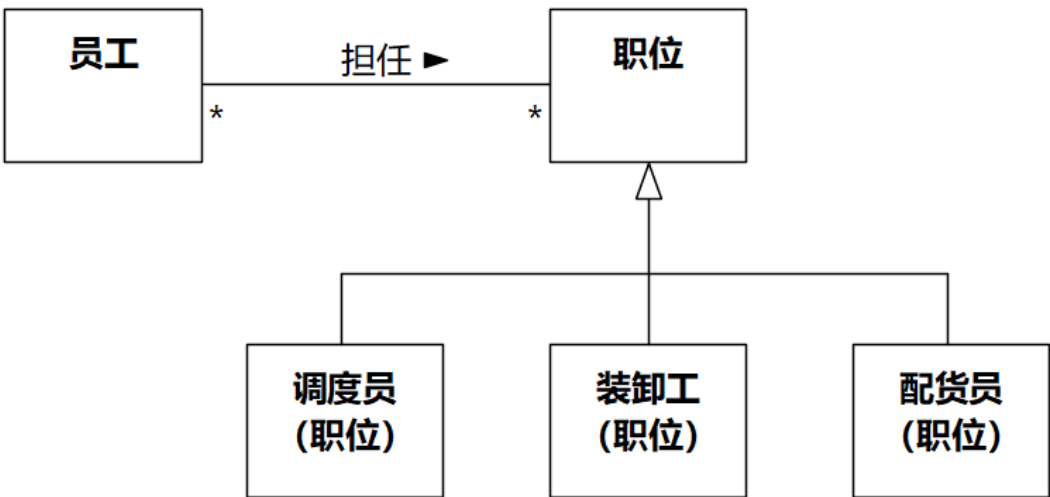


图 8-175 调整泛化关系的位置

图 8-175 中，“调度员”、“装卸工”、“配货员”的概念和图 8-173

相比含义已经不同，它们变成了“职位”的一种。“职位”的任意子集之间的交集应该为空。

如果“调度员”、“装卸工”、“配货员”取“职位”的概念，那么类似图 8-173 的图 8-176 其实也不是不可以。

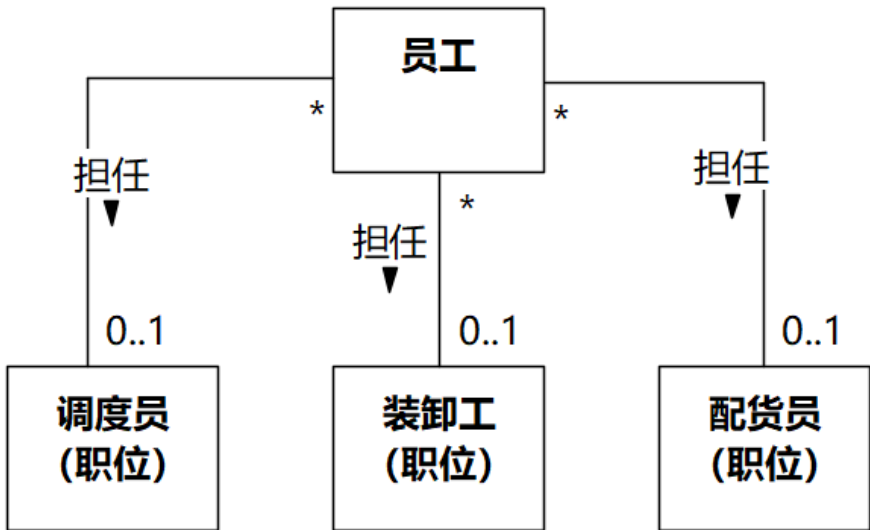


图 8-176 员工关联到具体职位

读者可以思考一下，如果把图 8-176 中的“员工”换成图 8-86 的“人”，下方的类换成“▲▲”和“○○”，可以吗？

如果对于不同职位，系统关注的差别在属性值上就可以体现，不需要在行为上体现，那么，图 8-175 的泛化关系可以取消，只保留“职位”，如图 8-177。



图 8-177 删去子类以及泛化关系

★图 8-175 到图 8-177 的“职位”，严格来说应该是“职位规格”。可以看得出来，“职位”实例的属性值不会随具体员工而变化。如果要记住张三做装卸工和配货员分别干了多少活，拿了多少绩效，还需要另外的类来区分。例如“员工-任职-职位”或“员工-职位-职位规格”。

(2) 完整

本书认为，不需要要求子类完整，否则会导致批量刷废话。

如图 8-178，一开始 A 有两个子类 B 和 C，知道应该是不完整的。如果为了完整，强行加一个口袋子类“其他 A”，那么很可能所有的泛化关系都会在结尾加一个“其他*”，这种批量的废话不如不要，缺省认为其不完整即可。

而且，“其他 A”的集合元素是不定的，现在的含义是“非 B 非 C”，假设以后 A 再加一个子类 D，“其他 A”的含义就变成“非 B 非 C 非 D”，元素也变少了。

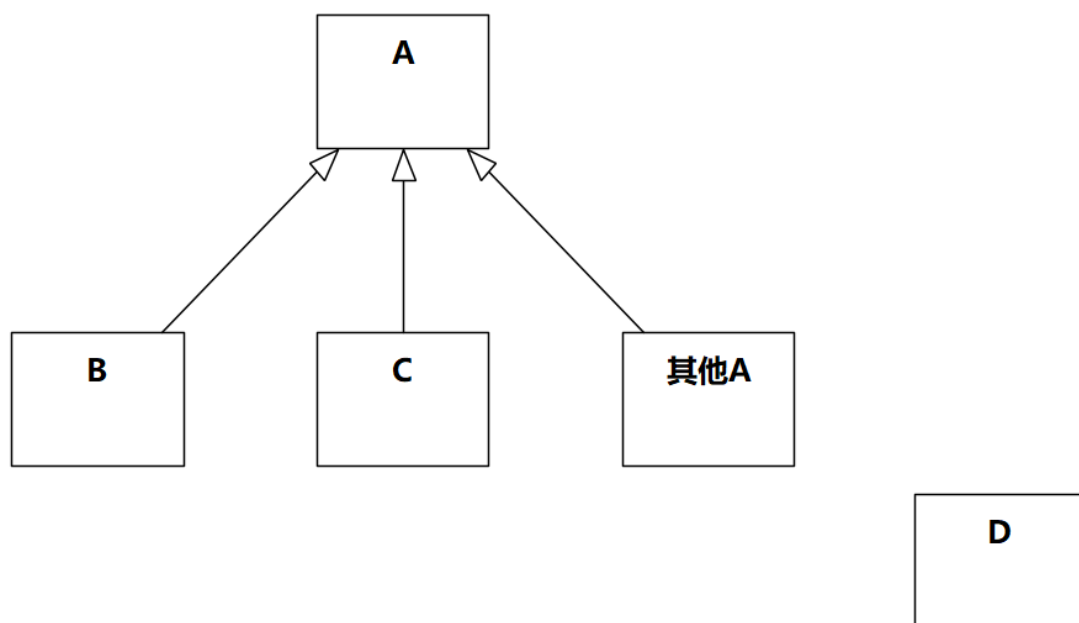


图 8-178 不要求完整

当然, 如果添加 D 为 B 或 C 更下一级的子类, 是可以的。如果 “其他 A” 经过时间的沉淀变成了一个领域术语, D 作为 “其他 A” 的子类也无妨——第 6 章所列的需求分类中的 “非功能需求” 就是这样的 “其他 A” 的例子。

8.3.5.2 Liskov 替换原则和矩形-正方形问题

1988 年, Liskov 在 “Data abstraction and hierarchy” 文章中提出了一个判断子类型的标准, 后来被称为 Liskov 替换原则 (LSP):

如果对于每个类型 S 的对象 o_1 , 都有类型 T 的对象 o_2 , 对于所有以 T 的形式定义的程序 P , 当用 o_1 替换 o_2 时, P 的行为不变, 那么 S 是 T 的子类型。

3.3. Type Hierarchy

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra. What is wanted here is something like the following substitution property [6]: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T . (See also [2, 17] for other work in this area.)

图 8-179 Liskov 文章截图

很多书和文章中提到 Liskov 替换原则时, 会以矩形和正方形 (有时会换成椭圆和圆) 的问题为例。

假设把正方形看作矩形的子类, 如图 8-180。

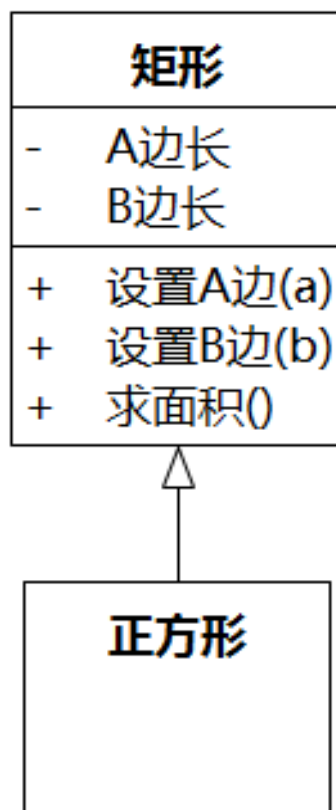


图 8-180 正方形作为矩形的子类

设置某矩形的 A 边长为 4, 再设置 B 边长为 5, 按照设想, 此时求面积应该得到 $4 \times 5 = 20$ 。如果用正方形代替矩形, 经过上面两次设置后, 最终得到的面积是 $5 \times 5 = 25$ 。根据 Liskov 替换原则判断, 图 8-180 不合适。

关于这个问题, 网络上搜索到的文章大多是从实现技巧的角度来解释和解决。

而从面向对象思想和领域逻辑的角度来思考和讨论这个问题, 上世纪 90 年代已经有不少文献, 讨论得较详细的有:

Bertrand Meyer 的书 "Object-Oriented Software Construction (2nd Edition)" (1998)

Kazimir Majorinc 的论文 “Ellipse-Circle Dilemma and Inverse Inheritance” (1998)

Walter L. Hiirsch 的论文 “Should Superclasses be Abstract?” (ECOOP '94)

UML 三友之一 James Rumbaugh 的书 “Object-Oriented Modeling and Design” (1991)、 “OMT Insights” (1996)。

本书从领域知识和集合的角度来谈一谈这个问题。

从领域知识上看，矩形的定义是：有一个角是直角的平行四边形。由此衍生的性质有：对边平行且相等、对角线互相平分且相等、面积=长×宽……。正方形也确实符合矩形的定义并具有矩形的性质，所以，正方形是矩形的子类（子集），是正确的。

但是，矩形还有其他没有画出来的子类（子集），而我们不知不觉地把没画出来的矩形子集的性质当成了所有矩形的性质，导致了冲突。

如图 8-181，A1 是 A 的子类，因为 A1 有公认的名字（例如正方形），所以被显式画出来。但 A 集合除了 A1 子集之外，还有其他子集，可能没有想好名字，没画出来。

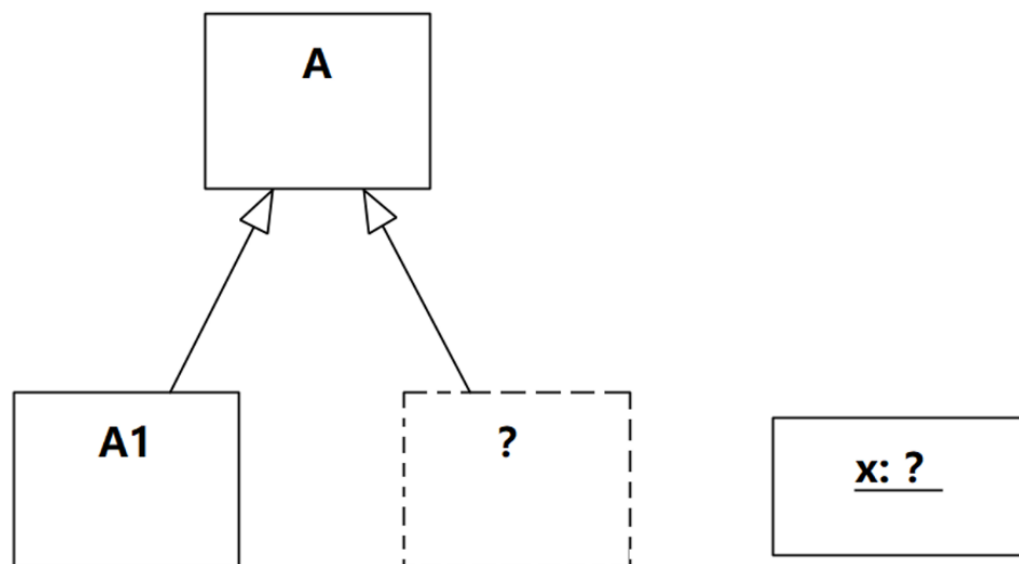


图 8-181 产生错觉的原因

假设此时我们想到一个对象 x , x 是 A 集合的一个元素, 但不是 $A1$ 集合的元素。也就是说, x 可能是图 8-181 中? (最大的? 即 $A-A1$) 的元素, 同时也是 A 的元素。

如果图上没有显式画出?, 只有 $A1$ 和 A , 既然 x 不属于 $A1$, 那就只剩下 A 了。于是, 我们在意识中记住了 “ x 是 A 的元素”, 忘记了和 $A1$ 同一级别的表述应该是 “ x 是? 的元素”, 然后就会产生? 和 A 相等的错觉。

正如上一小节所说, 在建模泛化关系时, 没有必要为了完整硬要加一个 “其他*” 什么之类的, 但我们心里必须有这个意识。

合适的处理方法是, 把不属于超类的性质从超类移除出去。

可以如图 8-182。正方形是矩形的子类, 除此之外, 还有其他子类, 如自由矩形、正方形和黄金分割矩形 (边长比为黄金分割比 0.618...: 1) 等。超类不实现 “设置 A 边(a)”、“设置 B 边(b)” 的操作, 也不应

先入为主地认为操作和属性会一一对应，例如“设置 A 边(a)”操作只会影响属性“A 边长”的值。

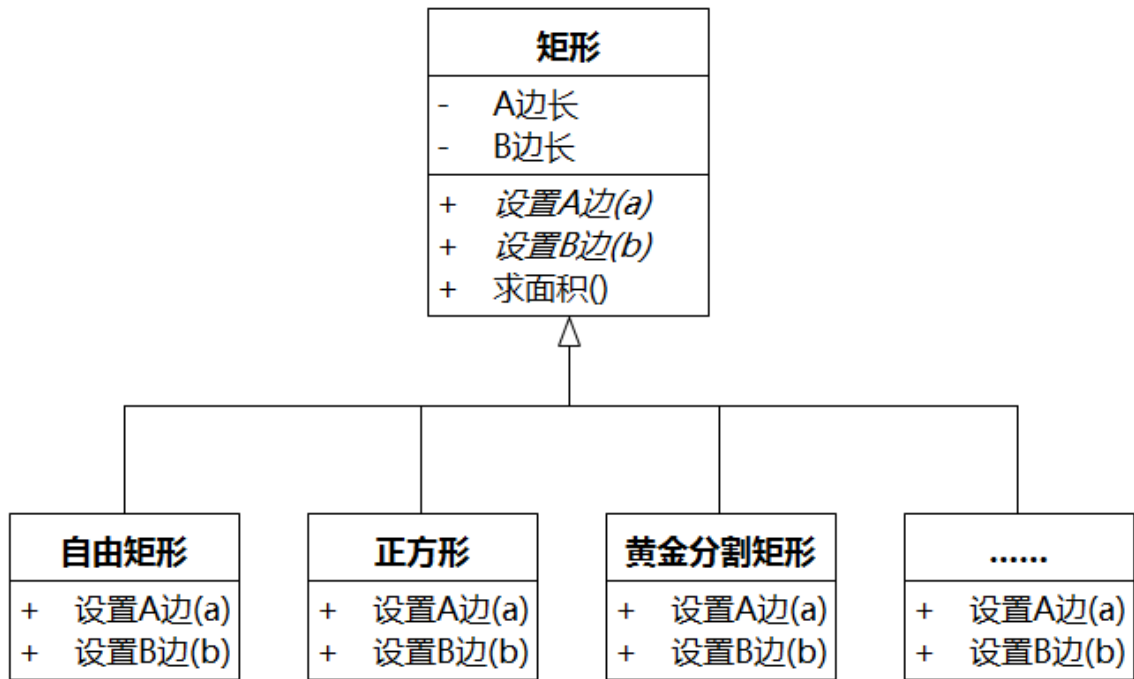


图 8-182 改进后的矩形泛化类图

更彻底的如图 8-183，把属性的定义也放到子类。

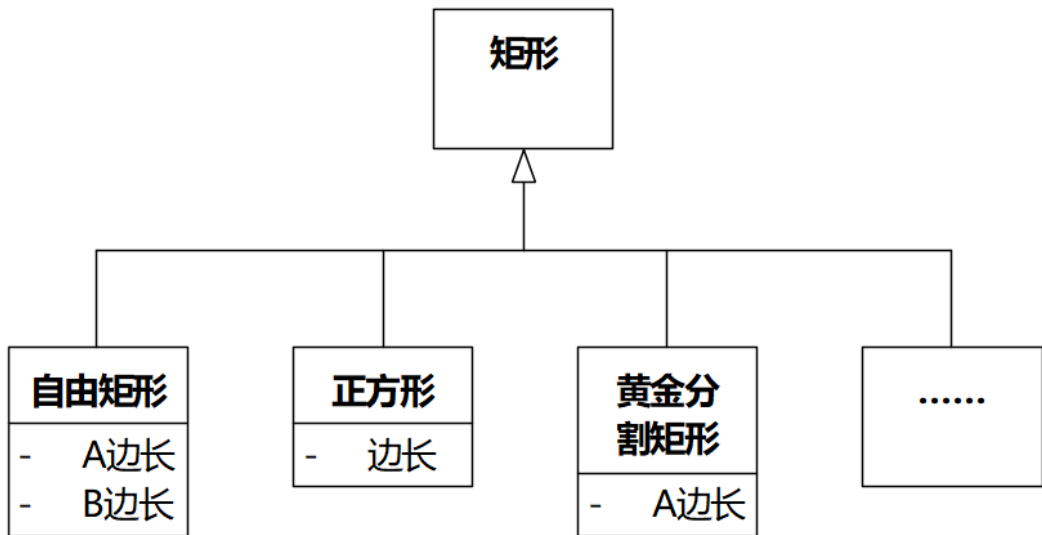


图 8-183 属性的定义放到子类

还有一个经常容易造成困惑的地方。

一个“自由矩形”对象，一开始的边长是 (3,4)，设置 A 边，让边长变成 (4,4)，它是不是就应该变成一个“正方形”对象？后面还能再自由设置边长吗？

它依然是“自由矩形”，只不过其两个边长在某个时刻碰巧相等，它并不是一个边长为 4 的“正方形”对象。这两个对象的属性值虽然相同，但遵循的行为规则是不同的。

类不止定义了属性，还定义了和这些属性相关的行为规则。

一把手牌 3 张 K，您觉得这个牌怎么样？



图 8-184 扑克牌手牌

如果是“斗地主牌”，算是好牌，如果是“21 点牌”，就已经爆了。

还有一种做法，把“等边”、“黄金分割”等都看成状态，如图 8-185。

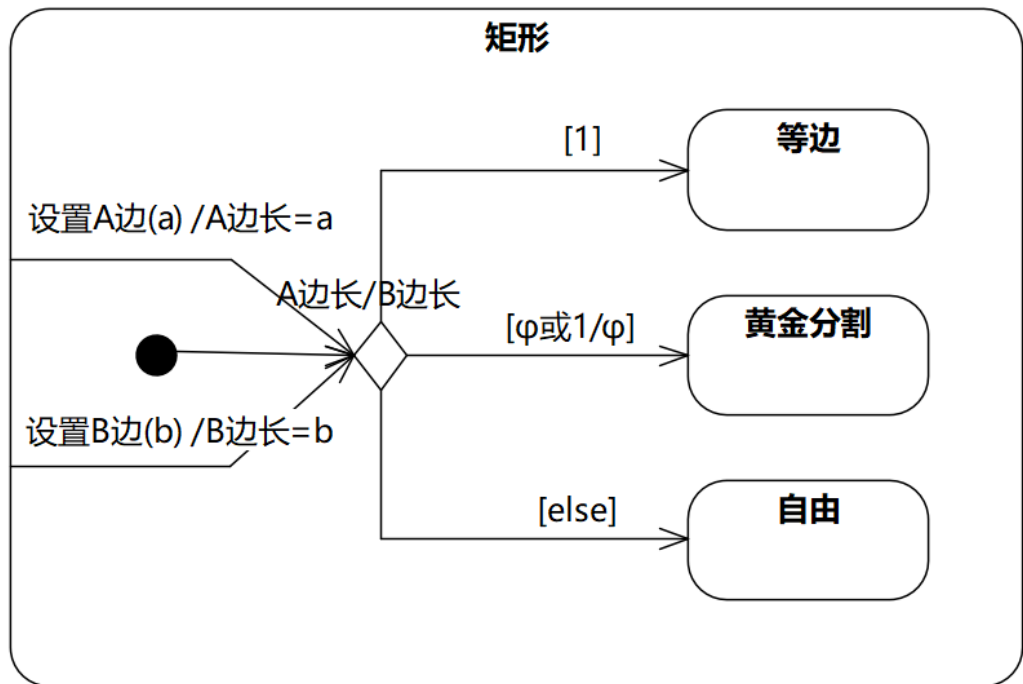


图 8-185 用状态来表达不同矩形

如果说一旦变成了正方形就永远只能正方形，可以如图 8-186。

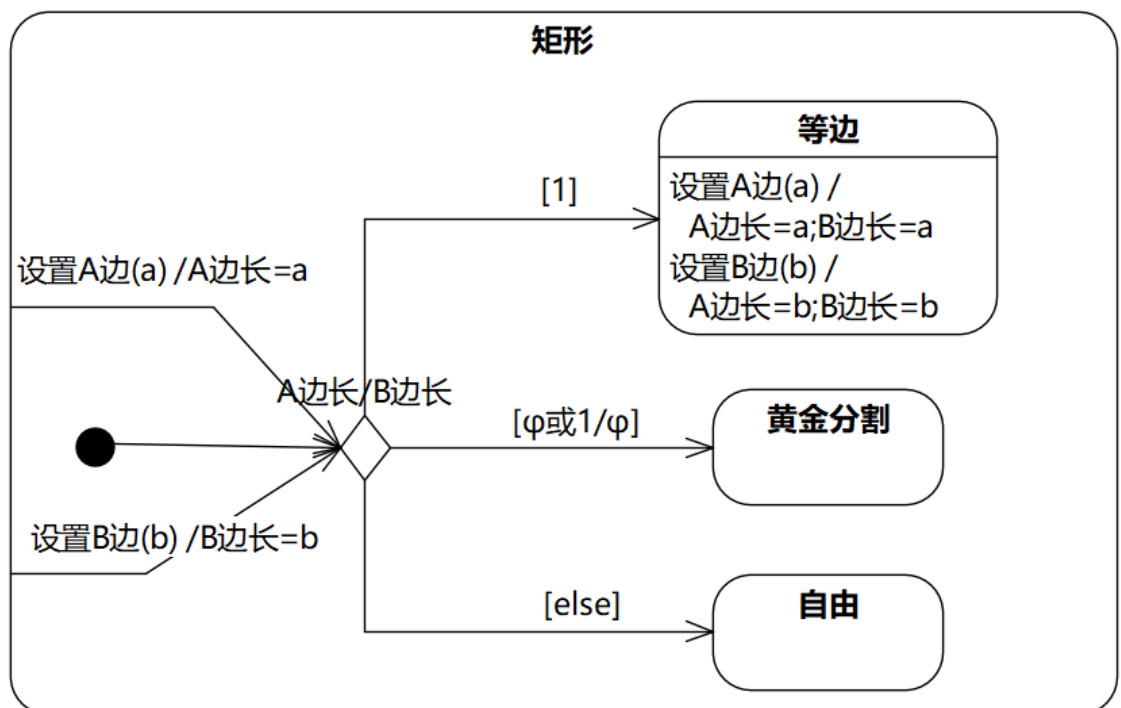


图 8-186 一旦正方形就只能正方形

第 9 章 分析 之 分析类图——案例篇

鸳鸯扣，宜结不宜解

《身似摇红烛影》，词：唐涤生，曲：王粤生，唱：红线女，1954

《软件方法（上）》以及下册 2018 年发布的电子版本，使用的案例是 “UMLChina 系统 2018”。案例中讨论了给特定分区的联系人发公开课通知邮件的领域逻辑，类图如图 9-1。

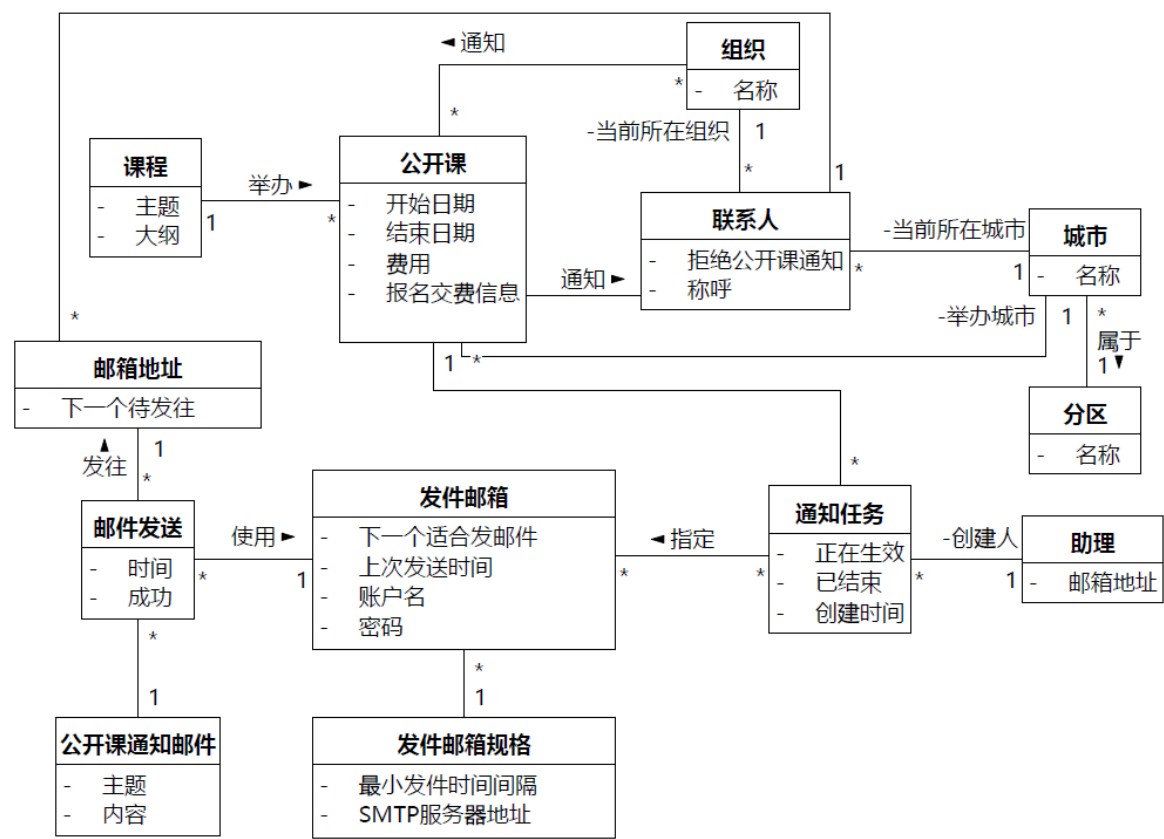


图 9-1 本书下册 2018 版本第 8 章的案例类图

时过境迁，原先使用邮件、短信甚至 QQ 的场合绝大部分已经改成使用微信。现在，UMLChina 已经很少使用发邮件和发短信的方式来通知学员，除了每年年底会发邮件询问最新邮政地址以便邮寄贺年卡。

而在微信上群发通知，从现在的礼仪来说，容忍度极低。所以，我们也不会针对微信的好友批量地发活动消息，只会在微信群里发。

第二个变化是，2019 年底持续至今的 2019-nCoV 疫情，使得公开课从线下转到了线上，不再有地域限制。

第三个变化是，自媒体平台（公众号、抖音、B 站等）成了首要的展示门户，自有网站的重要性大幅下降。

以上提到的是这几年周边业务环境的变化，当然，作为讲解建模的案例，这是小问题，无非是案例的核心域知识有些跟不上时代。

更大的问题是，上册所描述的举办公开课和群发邮件，涉及到的更多是查询的逻辑，如果是讲解类图的建模，是够用了，但状态变化涉及不多，后文讲解状态机图和序列图时不好展开，而通过状态机来封装修改属性值的逻辑是面向对象建模的重点。

考虑到以上变化，本书下册更换了案例。

案例一是写书时正在关注的另一个 UMLChina 流程：上课时做题并抽奖，涉及到考试和抽奖的领域知识。

案例二是我们正在研发的一款封装《软件方法》知识的智能建模工具——暂时命名为“发糕”，涉及到软件开发方法学的领域知识。

已成文的 2018 版本第 8 章的“UMLChina 系统 2018”案例剖析，继续作为参考案例保留，放在附录中。

9.1 案例一：答题抽奖

9.1.1 背景知识

UMLChina 的训练中，老师每讲解一个知识点，就会用一个做题软件让学员做一些题目，考查学员是否掌握。这些题目由老师在课前设置好。为了提高学员做题的兴趣，老师会用一个抽奖软件为答对题目的学员抽奖。

具体过程如下：

(1) 老师请求做题软件显示下一道要做的题目，然后等待 10-60 秒，让学员阅读题目。

*同样难度的题甚至是同一道题，老师目前是凭感觉留给学员阅读题目的时间，留多了会造成浪费，留少了学员会觉得不公平。

软件开发中需求工作的目的是

- A) 更好地指导设计
- B) 满足软件工程需求规范
- C) 对系统做概要的描述
- D) 让系统更加好卖



图 9-2 答题抽奖流程现状-1：出题

(2) 老师请求抽奖软件随机抽取一名学员，抽奖软件随机抽取并显

示抽中学员名字，老师在教室里寻找该学员所在位置，点名该学员回答问题。



图 9-3 答题抽奖流程现状-2：点名

(3) 学员思考并口头回答问题，老师确认听清学员的回答后（人多时噪音较多），向做题软件提交学员的回答，做题软件判断并反馈对错。

*有的学员思考时间过长，甚至有学员的答题习惯是先把题目念一遍，导致老师不得不出声催促。

软件开发中需求工作的目的是

- ☒ A) 更好地指导设计
- ☐ B) 满足软件工程需求规范
- ☐ C) 对系统做概要的描述
- ☐ D) 让系统更加好卖

答错，年轻人不讲武德

软件开发中需求工作的目的是

- A) 更好地指导设计
- B) 满足软件工程需求规范
- C) 对系统做概要的描述
- D) 让系统更加好卖

答对，个个都是人才



图 9-4 答题抽奖流程现状-3：判对错

(4) 如果学员答对，老师请求抽奖软件为学员抽奖，抽奖软件从当前奖池中随机抽取奖品，将抽中的奖品从奖池扣除，反馈抽中的奖品信息，更新剩余奖品数量，更新学员答对排行榜。奖品可能是现金（金额从 1.28 元到 40.96 元），也可能是实物（书、饮料、小食等），还有一定比例的奖品是“木有”-没有抽到奖品。



图 9-5 答题抽奖流程现状-4：抽奖

(5) 如果奖品是实物，老师将实物发给学员；如果奖品是现金，老师通过手机上的微信发红包给学员。



图 9-6 答题抽奖流程现状-5：发红包

9.1.2 愿景

目标系统：

UMLChina 系统 2022

目标组织：

UMLChina

老大：

UMLChina 负责人 潘加宇

需改进指标：

UMLChina 训练中，花费在回答问题和抽奖上的平均时间

指标当前值：

3 分钟/题

指标改进值:

2 分钟/题

9.1.3 业务序列图

针对以上流程，绘制现状的业务序列图如图 9-7。

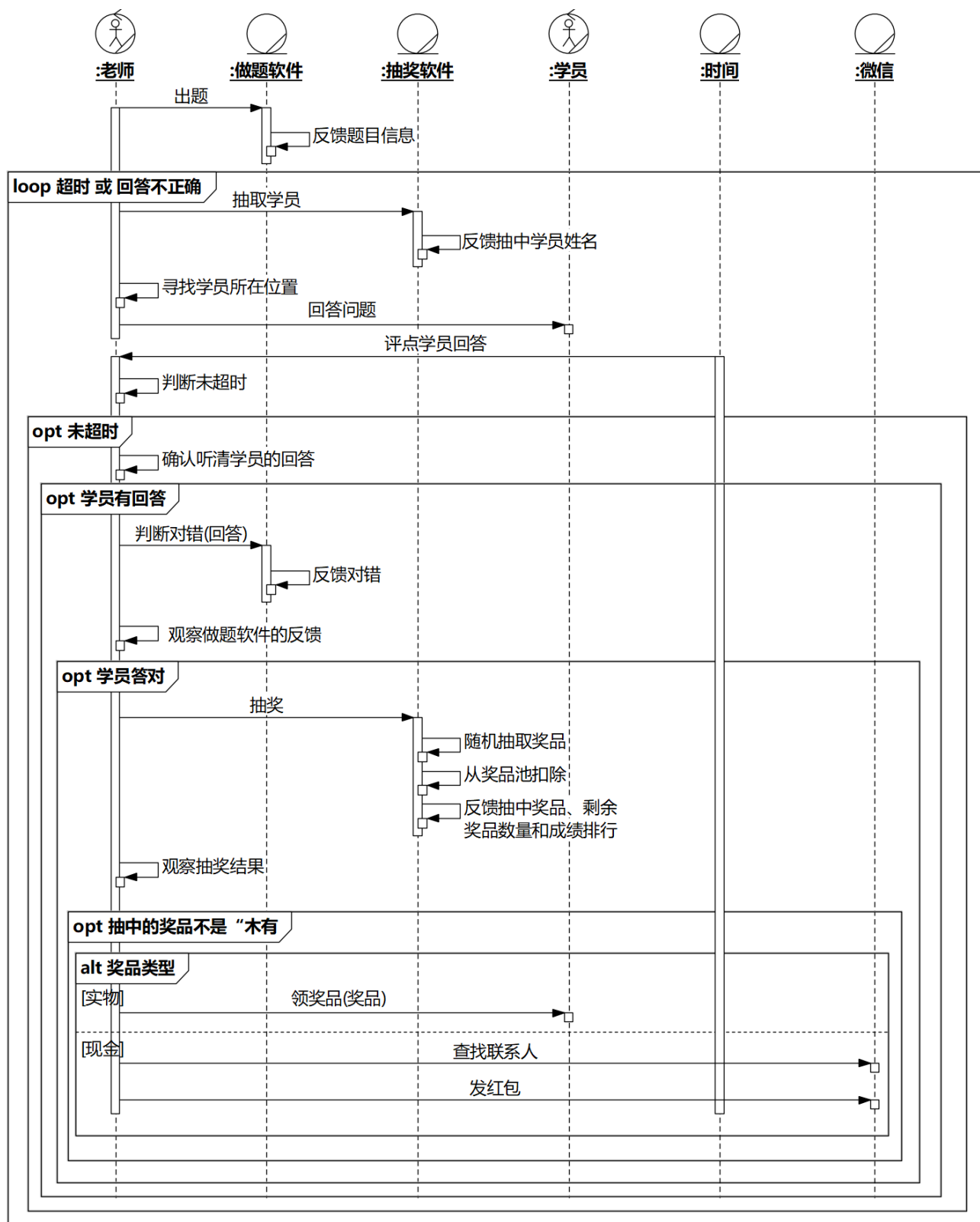


图 9-7 答题抽奖流程现状业务序列图

从图 9-7 可以看到，做题软件、抽奖软件和微信之间不直接通信。老师担任了信息搬运工的角色，用自己的手指按动电脑上的鼠标，把学

员的回答搬运到做题软件里，用自己的手指按动手机上的软键盘，把做题软件抽出来的奖品金额搬运到微信里……。这种情况符合《软件方法》第 4 章提到的“改进模式二：改善信息流转”。

“寻找学员所在位置”、“判断超时”等逻辑封装在人脑中。这种情况符合《软件方法》第 4 章提到的“改进模式三：封装领域逻辑”。

改进的业务序列图如图 9-8。

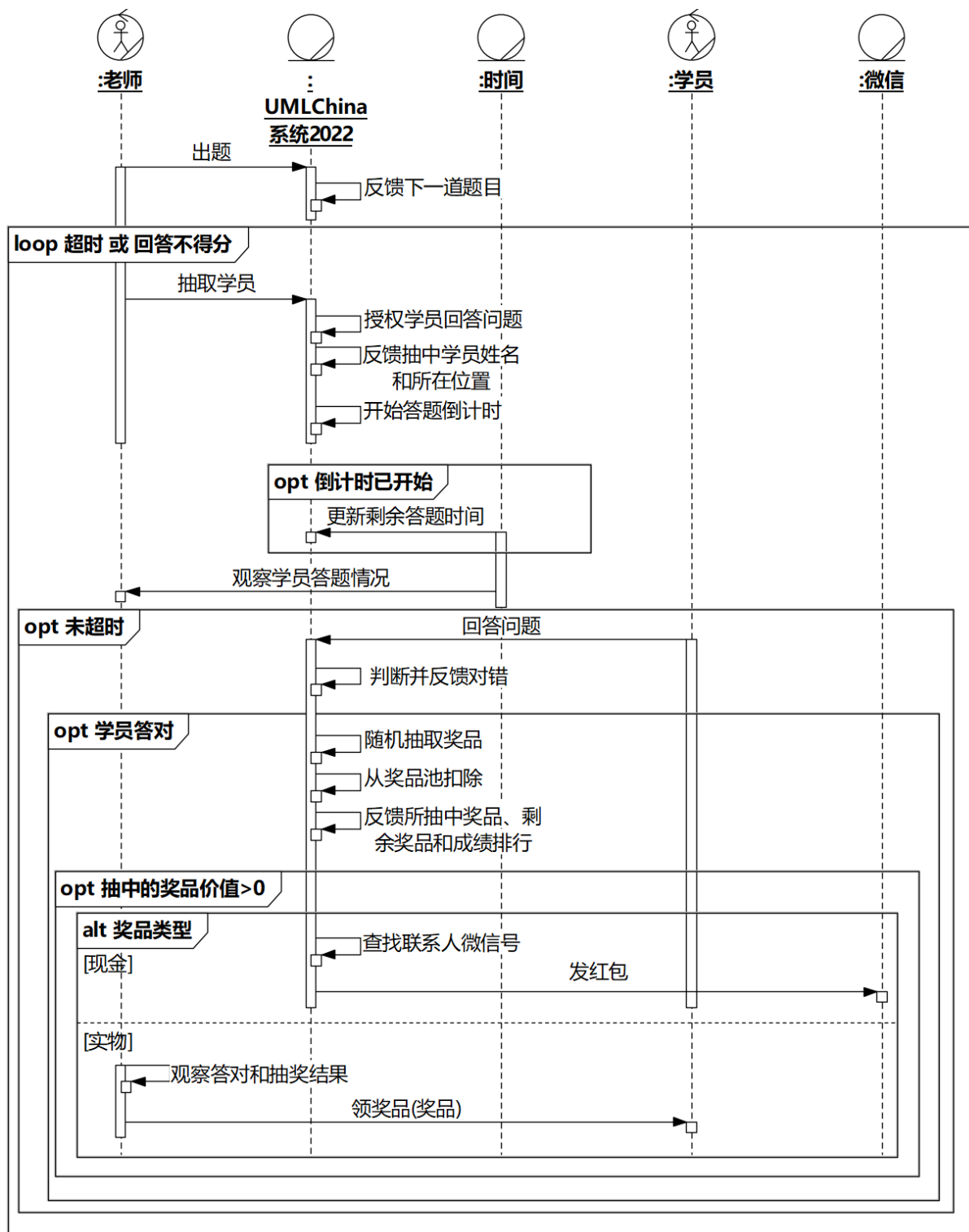


图 9-8 答题抽奖流程改进业务序列图

注意，引进的系统依然叫“UMLChina 系统”，并非“UMLChina 答题抽奖系统”。“答题抽奖”可以看作由若干用例组成的用例集。

实际上，现实中的“UMLChina 系统”现在已经有了不少类，例

如“学员”、“课程”，但为了讲解更清晰，书中假设不存在现有的类，从头构思“答题抽奖”案例用例集所需的类。

但要提醒的是，这样做的原因只是为了本书讲解方便，这是一个盘外因素，和系统内部组件的耦合内聚并不相关。

近几年，“微服务”之风盛行，但该不该切分为微服务，如果一定要切分应如何切分，是一个非常复杂的问题。有些人未必乐意面对以及学习如何应对这些复杂问题，于是就冒出来一些投资少、见效快、产量足的“微服务伪创新”，它们避开了复杂的软件结构问题，专从盘外招下功夫。

就像第 1 章所言，判断地震来了哪座楼倒塌的可能性最小，需要考虑大楼的结构、所用的材料、所在位置的地质环境等，但这涉及到艰深的工程力学、材料学和地质学知识。于是有人发明了简单易行的办法，从工作服的颜色、工人是否结对洗澡、施工队开会时是否站立等方面来找原因。

之所以做以上提醒，是怕读者误会，以为作者推荐从愿景、业务序列图推导出的用例集作为划分“微服务”的依据。

如何“切分微服务”，更好的做法，参见本书第 8 章之后各章的知识讲解。

9.1.4 系统用例图

从图 9-8 映射“UMLChina 系统 2022”的用例图如图 9-9。

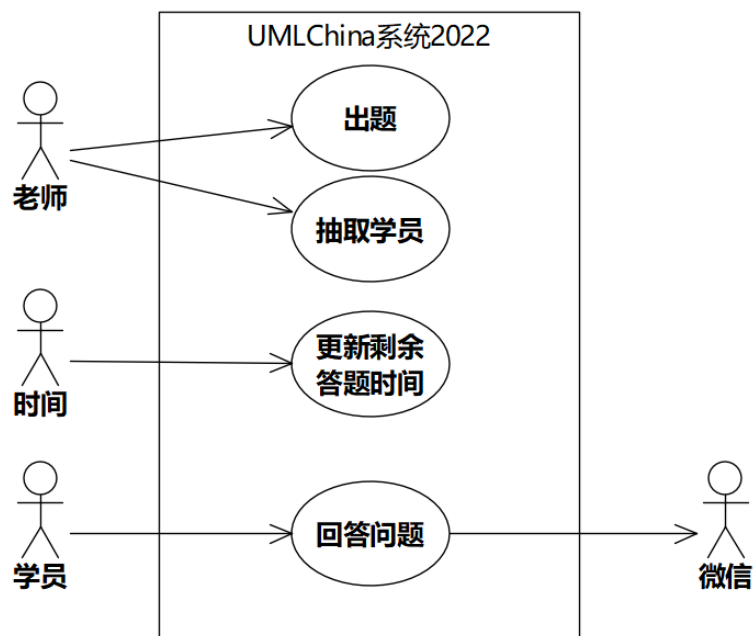


图 9-9 “UMLChina 系统 2022” 的用例图

9.1.5 系统用例规约

“学员→回答问题” 的用例规约如下：

用例名：

回答问题

执行者：

学员（主）、微信（辅）

涉众利益：

老师-担心评价更新不及时，让学员觉得陈旧。

学员-担心自己碰到有陷阱的题目，导致周围同事特别是领导认为自己答不对简单的题目，能力不行。

学员-担心题目字太小看不清楚，担心手滑选错答案。

老师-担心抽奖结果被认为是预设的，导致学员失去兴趣。

老师-担心大奖出现得太快，导致后面的学员没有动力。

学员-担心抽奖不公平，轮到自己抽奖时运气太差，轮到别人抽奖时运气太好。

奖品生产厂商负责人-担心自己的产品被用作不好的用途。

微信系统管理员-担心自己维护的系统受影响发生故障。

基本路径：

1. 学员提交回答。
2. 系统验证回答有效。
3. 系统判分，保存判分结果。
4. 系统反馈判分结果。
5. 系统验证得分达到抽奖要求。
6. 系统随机抽取奖品，将抽中的奖品从奖池移除，保存抽奖结果。
7. 系统反馈抽奖结果、剩余奖品和学员成绩排行。
8. 系统验证抽中的奖品为现金类型且存在学员的微信号。
9. 系统请求微信向学员的微信号发红包。
10. 系统保存发奖结果。
11. 系统反馈发奖结果。

扩展：

2a. 回答无效:

2a1. 系统反馈回答无效。

2a2. 用例结束。

字段列表和业务规则:

1. 回答选择了题目的某个或若干个选项。

*系统暂时不考虑选择题之外的题型。

2. 有效规则: 回答选择的选项必须属于同一道题目; 单选题的回答只能是 1 个选项, 多选题的回答至少有 2 个选项。

3. 判分结果=学员+回答选项集+得分。

3. 判分规则: 所回答选项集和题目预设正确选项集完全相同, 则得到该题全部分值, 否则该题得分为 0。

*分值: 某个知识点会有一组题目让学员回答, 同一组题目中, 可根据题目难度为题目设置不同的分值, 难度越大, 分值越高。难度是相对的, 同一道题目, 放在 A 组可能属于难题, 放在 B 组可能就相对容易。

4. 判分结果=学员姓名+得分+评价。

4. 评价从得分值所在的分值区间适用的评价集合中随机抽取。

*评价: 一些用于活跃气氛的热门用语, 和学员回答的得分值对应, 一个得分值会准备好多条评价。例如, 得 0 分, 评价可能有“不讲武德”、“耗子尾汁”、“你站在此处不要动”等, 得 1 分, 评价可能有“打工是不可能打工的”、“个个都是人才”等, 得更高分, 评价可能有“人类高

质量男（女）性”、“yyds”等。

5. 抽奖要求：得分值大于 0。

6. 抽奖结果=学员+奖品+抽奖时间。

6. 抽奖规则：抽奖时，如果奖池有多于一件奖品，而且价值不完全相同，那么把价值最大的奖品抽中的概率调低为其他奖品的一半。

7. 抽奖结果=奖品名称。

7. 剩余奖品=奖品名称+剩余数量。按奖品的价值降序排序。

*价值：每种奖品会设置一个价值，现金的价值为现金的金额，实物的价值为该实物的估值，未抽到奖励视为抽到价值为 0 的奖品。

7. 成绩排行=学员姓名+成绩+中奖次数，先按成绩降序排序，再按中奖次数降序排序，最后按学员姓名升序排序。

*中奖：抽到的奖品价值大于 0，为中奖。

7. 学员成绩为当前活动中，学员所提交回答的得分总和。

*某一次公开课或内训称为一次活动，一次活动可能会持续几天，学员会一直参加。在活动过程中，老师会出很多组试题（或者说出很多张试卷）。

*学员可以参加很多次 UMLChina 的活动。例如，两个月前参加了一次“软件需求设计方法学全程实例剖析”培训，这个月又参加“业务建模和需求高阶”培训。此处的学员成绩指在单个活动内的成绩，不跨活动累加。

9. 输入输出信息参见最新版微信支付接口规则。

9. 红包金额为现金奖品的价值。

10. 回答+是否发放成功+发放时间。

11. 是否发放成功。

质量需求 and 设计约束：

1. 针对初次使用本用例的健康人样本，要求平均每秒 1 次按指令提交回答 20 次无错误。通过的比例不少于 99%。

6. 本步骤的执行时间控制在 3-5 秒之间。

*即使计算时间很短，也不宜太早出奖品，因为会让学员觉得没有在随机抽奖，而是预设好的结果。

9.1.6 分析类图

从“学员→回答问题”用例规约的基本路径到业务规则部分提炼。我们给关键词加下划线，逐个分析。

注意：加下划线只是为了在书中讲解方便，实际建模工作中不需要这样做，只需要看着需求规约或其他素材，一点点提炼。

9.1.6.1 步骤 1 及其补充约束

用例规约：

1. 学员提交回答。

1. 回答选择了题目的某个或若干个选项。

提炼：

***学员**

这里的“学员”是执行者，指外面那个和系统交互的人肉系统“学员”，而不是系统内部维护学员相关信息和规则的“学员”类——有没有这个类，目前还不能判断。

执行者映射的是边界类，如图 9-10：

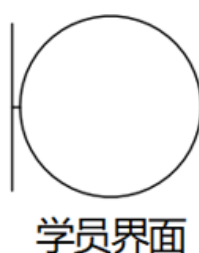


图 9-10 执行者映射边界类

***回答**

“回答”提炼为类；

***回答选择了题目的某个或若干个选项。**

“题目”、“选项”提炼为类；

“回答”关联“选项”，关联名称可以叫“选择”，“回答”和“选项”的多重性均为*；

“题目”关联“选项”，“题目”的多重性为 1，“选项”的多重性为*。

在没有足够证据时，关联一律表示为普通关联。

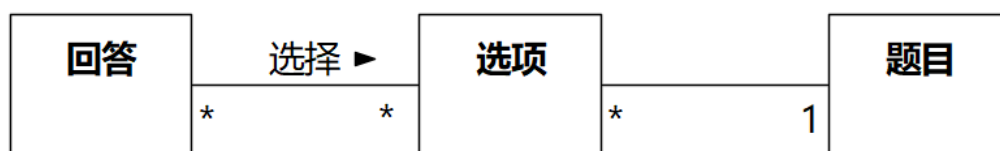


图 9-11 回答、选项和题目

9.1.6.2 步骤 2 及其补充约束

用例规约：

2. 系统验证回答有效。

2. 有效规则：回答选择的选项必须属于同一道题目；单选题的回答只能是 1 个选项，多选题的回答至少有 2 个选项。

提炼：

*系统

如第 8 章知识点所述，“系统”不识别。

*有效

“有效”是一个形容词，可以作为某个类的状态。此处“有效”描述的是“回答”，映射为“回答”类的“状态属性”。后面深入建模时，会通过状态机中的状态来取代这些“状态属性”。

*回答选择的选项必须属于同一道题目

图 9-11 无法对此做出约束，需要对“回答”加一个约束，例如：

```
(from 选项 in 选项 s select 选项.题目).Distinct().Count() <= 1
```

*单选题的回答只能是 1 个选项，多选题的回答至少有 2 个选项

“单选题”、“多选题” 可以提炼为类，作为“题目” 的子类，如图 9-12。

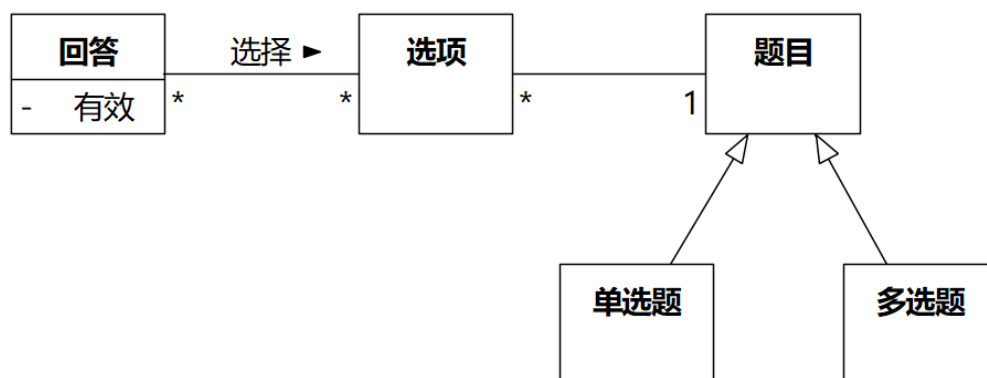


图 9-12 单选题和多选题

提炼“只能是 1 个选项”、“至少有 2 个选项”背后的领域概念——“回答选项个数下限”、“回答选项个数上限”，把 1、2 等具体数字消除掉。

“回答选项个数下限”、“回答选项个数上限”是哪个类的属性呢？如果作为“题目”或“单选题”、“多选题”的属性，会导致有很多“题目”对象的这两个属性值相同。

可以应用第 8 章讲述的泛化转换为关联的知识，添加一个“类类型”类叫“题型”，同时去掉泛化关系。“回答选项个数下限”、“回答选项个数上限”作为“题型”的属性。

“题目”关联“题型”，“题目”的多重性为*，“题型”的多重性为 1。

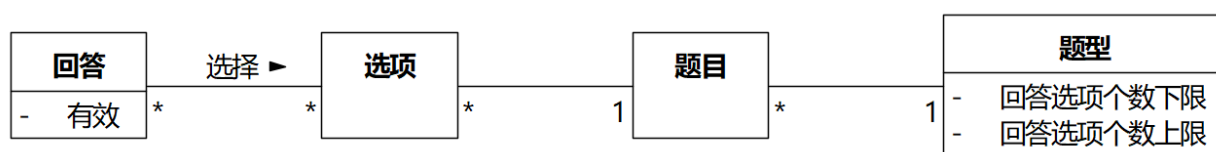


图 9-13 题型

“单选题的回答只能是 1 个选项,多选题的回答至少有 2 个选项”
这条关于“有效”的规则,可以表达成“回答”的一个约束,例如:

选项 s.Count>=选项 s.FirstOrDefault().题目.题型.回答选项个数
下限 && 选项 s.Count<=选项 s. FirstOrDefault().题目.题型.回答选
项个数上限

这个约束以及上面的约束,都会体现在“回答”的行为中。后面我们还会再讨论。

9.1.6.3 步骤 3 及其补充约束

用例规约:

3. 系统判分, 保存判分结果。

3. 判分结果=学员+回答选项集+得分。

3. 判分规则: 所回答选项集和题目预设正确选项集完全相同, 则得到该题全部分值, 否则该题得分为 0。

*分值: 某个知识点会有一组题目让学员回答, 同一组题目中, 可根据题目难度为题目设置不同的分值, 难度越大, 分值越高。难度是相对的, 同一道题目, 放在 A 组可能属于难题, 放在 B 组可能就相对容易。

提炼:

*学员

“学员”提炼为一个类, 和“回答”关联, “学员”多重性为 1,

“回答” 多重性为*。

*得分

提炼为 “回答” 的属性。

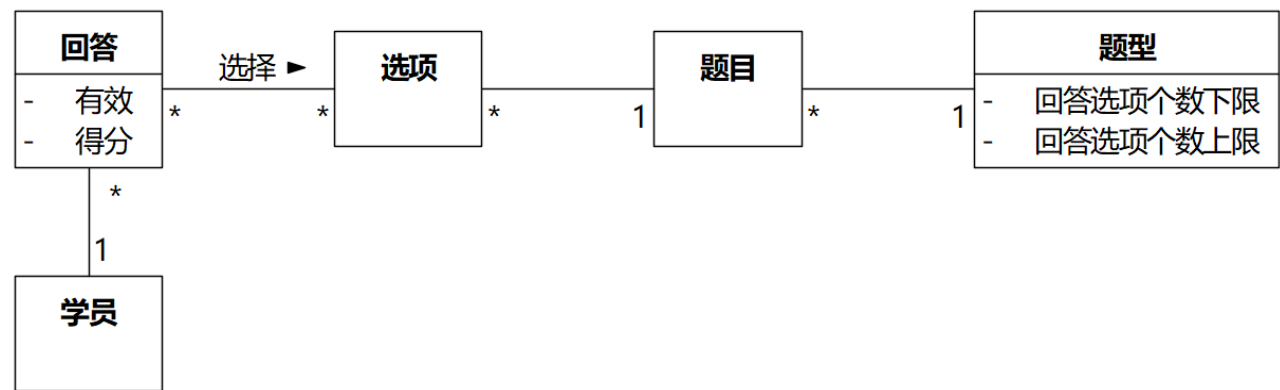


图 9-14 添加 “学员”

*题目预设正确选项集

系统需要记住某道题的哪些选项是答案，可以给 “选项” 增加一个属性 “正确”。

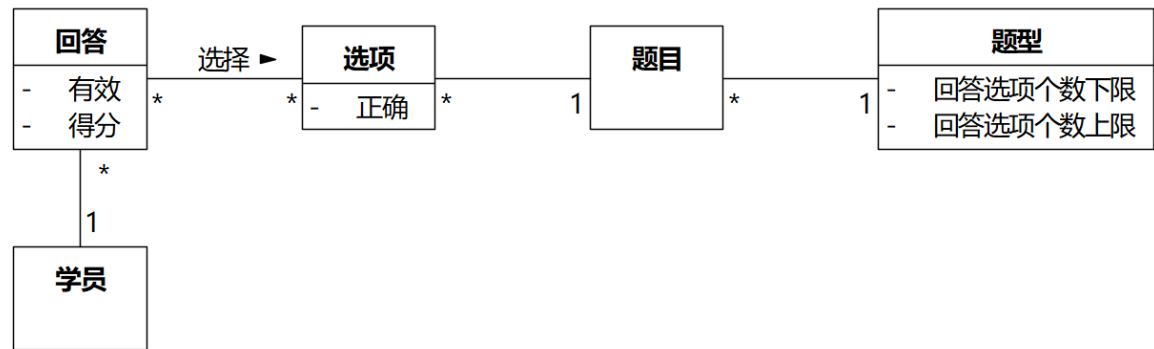


图 9-15 给选项添加 “正确” 属性

注意，这种情况下，“选项” 和 “题目” 的关联中，“选项” 只能属于一道 “题目”。如果可以多个 “题目” 共享 “选项”，那么 “正确” 的值将无法确定。

如果一定要共享“选项”的内容，可以增加一个类“命题”（或“陈述”），把“内容”作为“命题”的属性。“选项”和“命题”关联，“选项”多重性为*，“命题”多重性为1。“选项”关联到的“命题”和“题目”决定“选项”的“正确”，（命题，题目）→正确。

例如，针对命题：“挂号”是用例。如果题目说“研究对象为医院”，那它作为选项有可能是错误选项，如果题目说“研究对象为医院信息系统”，那它作为选项有可能是正确选项。更直接的例子，问“以下选项正确的是”和“以下选项错误的是”时，选项是否正确刚好是颠倒的。

此时模型应该是这样：

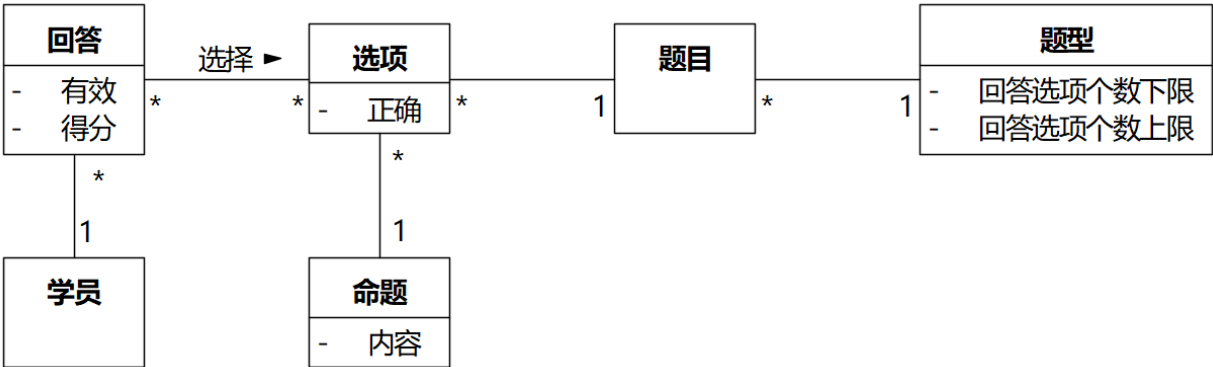


图 9-16 另一种选择

可能有人会想到图 9-17 两个关联的做法，不过这个做法并不合适。“正确”是由（命题，题目）的关联决定的，正确选项必须是选项的子集。如果像图 9-17 这样使用两个独立的关联，就有可能出现某道“题目”的“选项”集合是{1,2,3,4}，而“正确选项”集合却是{5,6}的矛盾。

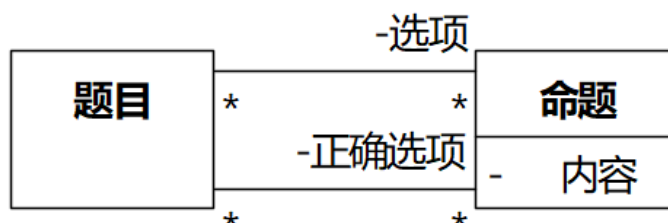


图 9-17 第三种选择

我们决定采用图 9-15 的做法，不需要“命题”类，“内容”和“正确”都作为类“选项”的属性。也就是说，不共享选项的内容，当多道题目的选项内容相同时，重复其内容。

*分值：某个知识点会有一组题目让学员回答，同一组题目中，可根据题目难度为题目设置不同的分值，难度越大，分值越高。难度是相对的，同一道题目，放在 A 组可能属于难题，放在 B 组可能就相对容易。

“分值”是谁的属性？从“该题全部分值”看似是“题目”的属性，但从“同一组题目中，可根据题目难度为题目设置不同的分值”又可以得知，“题目”不能决定“分值”，而是（题目组，题目）→分值，需要一个能决定（题目组，题目）的类，我们把这个类叫“试题”，题目组可以称为“试卷”。

虽然案例中的场景和常见的考试比起来，没那么正式也没那么严肃，但本质上是一致的。我们这个考试有奖品，正式的考试奖品更大，可能是奖学金，可能是升学资格，可能是编制。

“试题”和“试卷”关联，“试题”的多重性为*，“试卷”的多重性为 1；“试题”和“题目”关联，“试题”的多重性为*，“题目”的多重性为 1。

重性为 1。

因为回答的得分和试题的分值有关,而一道题目可以对应多道试题,通过原有的“回答→选项→题目”无法导航到特定的试题。

这个时候,有必要把“回答”和“试题”关联,“回答”的多重性是*,“试题”的多重性是 1。

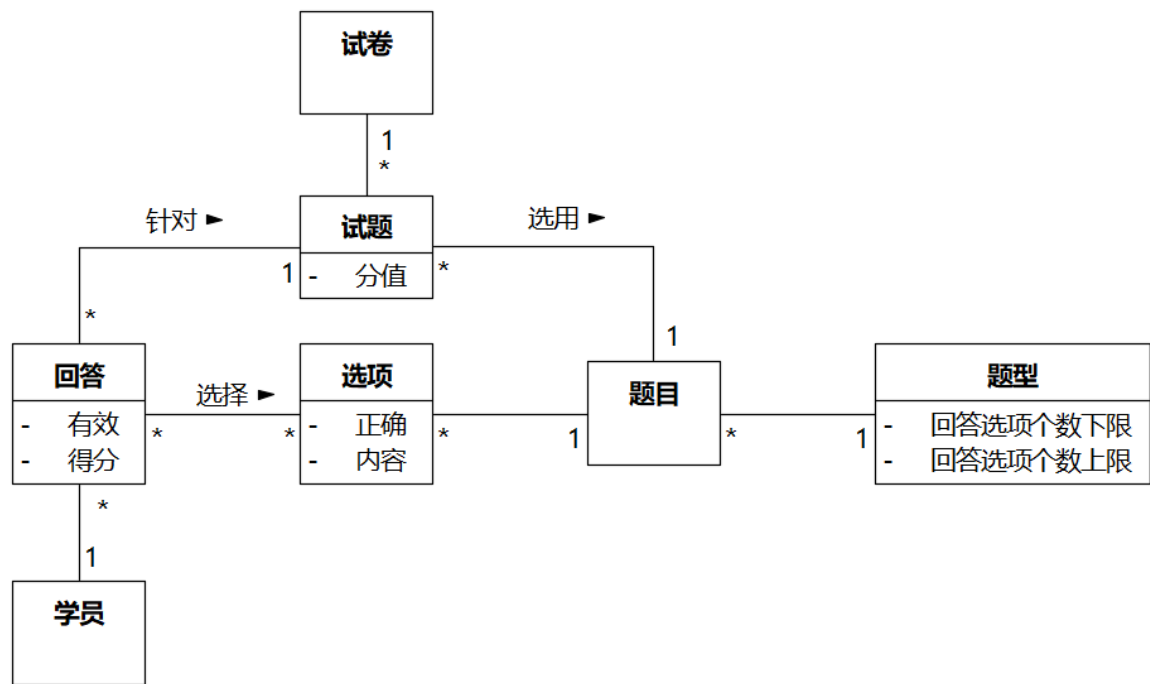


图 9-18 试题

之前关于回答有效的约束表达式也会发生变化:

//选项最多只属于一道题

(from 选项 in 选项 s select 选项.题目).Distinct().Count()<=1

//选项的题目和试题的题目是同一道题

选项 s.FirstOrDefault().题目==试题.题目

//选项的个数符合要求

选项 s.Count>=试题.题目.题型.回答选项个数下限 && 选项
s.Count<=试题.题目.题型.回答选项个数上限

//判断是否答对，即选项集等于题目正确选项集

选项 s Equal (试题.题目.选项 s.Where(选项 => 选项.正确))

9.1.6.4 步骤 4 及其补充约束

用例规约：

4. 判分结果=学员姓名+得分+评价。

4. 评价从得分值所在的分值区间适用的评价集合中随机抽取。

*评价是一些用于活跃气氛的热门用语，用于评价学员的得分。例如：

得 0 分，评价可能有“抛开事实不谈”、“不讲武德”、“耗子尾汁”、“你站在此处不要动”等；

得 1-2 分，评价可能有“打工是不可能打工的”、“个个都是人才”、“维持一下生活这样子”等；

得 3-4 分，评价可能有“下赛季你防詹姆斯”、“人类高质量男（女）性”、“yyds”等。

提炼：

*姓名

提炼为“学员”的属性。

*评价

提炼为类 “评价”。

*分值区间

提炼为类 “分值区间”，属性为 “下限” 和 “上限”。“分值区间” 和 “评价” 关联，“分值区间” 的多重性为 1，“评价” 的多重性为*。

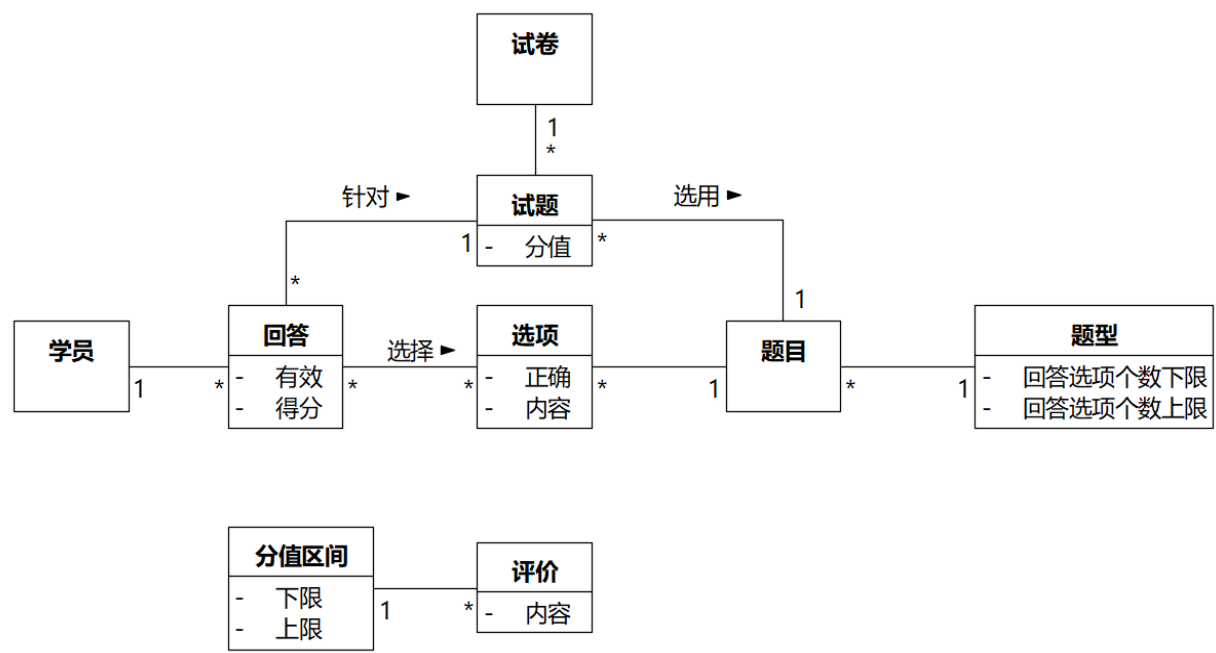


图 9-19 分值区间和评价

9.1.6.5 步骤 5 及其补充约束

用例规约：

- 5. 系统验证得分达到抽奖要求。
- 5. 抽奖要求：得分值大于 0。

提炼：

这一步没有新的类和属性需要提炼。

9.1.6.6 步骤 6 及其补充约束

用例规约:

6. 系统随机抽取奖品, 将抽中的奖品从奖池移除, 保存抽奖结果。

6. 抽奖结果=学员+奖品+抽奖时间。

6. 抽奖规则: 抽奖时, 如果奖池有多于一件奖品, 而且价值不完全相同, 那么把价值最大的奖品抽中的概率调低为其他奖品的一半。

提炼:

*奖品

提炼类 “奖品”。

*奖池

提炼类 “奖池”。“奖池” 和 “奖品” 关联, “奖池” 多重性为 1, “奖品” 多重性为*。

*抽奖结果

提炼类 “抽奖”。“抽奖” 和 “奖品” 关联, “抽奖” 多重性为 1, “奖品” 多重性为 0..1。“抽奖” 和 “回答” 关联, “抽奖” 多重性为 0..1, “回答” 多重性为 1。

*抽奖时间

提炼为 “抽奖” 的属性 “时间”。

*价值

提炼为 “奖品” 的属性。

*概率

这是若干类和属性计算得到的结果，不需要提炼为类和属性。

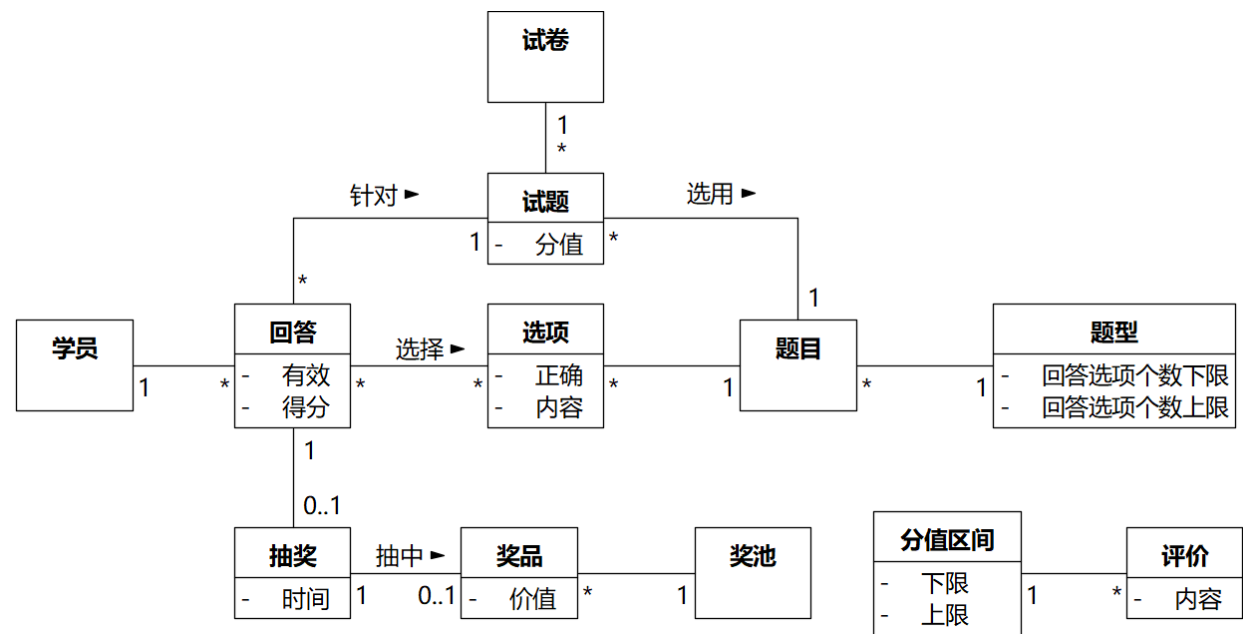


图 9-20 抽奖

9.1.6.7 步骤 7 及其补充约束

用例规约：

7. 系统反馈抽奖结果、剩余奖品和学员成绩排行。

7. 抽奖结果=奖品名称。

7. 剩余奖品=奖品名称+剩余数量。按奖品的价值降序排序。

*价值：每种奖品会设置一个价值，现金的价值为现金的金额，实物的价值为该实物的估值，未抽到奖励视为抽到价值为 0 的奖品。

7. 成绩排行=学员姓名+成绩+中奖次数，先按成绩降序排序，再按中奖次数降序排序，最后按学员姓名升序排序。

*中奖：抽到的奖品价值大于 0，为中奖。

7. 学员成绩为当前活动中，学员所提交回答的得分总和。

*某一次公开课或内训称为一次活动，一次活动可能会持续几天。在活动过程中，老师会出很多组试题（或者说，出很多张试卷），然后针对每道题点名学员回答问题。

*学员可以参加很多次活动。例如，两个月前参加了一次“软件需求设计方法学全程实例剖析”培训，这个月又参加“业务建模和需求高阶”培训。此处的学员成绩指在单个活动内的成绩，不跨活动累加。

提炼：

*剩余奖品、剩余数量

这两个概念如何处理，要看“奖品”如何定义。可以把“剩余数量”作为某个类的属性，也可以通过计算“奖池”关联的“奖品”对象数量得到。此处暂时放下，等到处理完其他概念后，我们再回来看这两个概念。

*奖品名称

给“奖品”加一个属性“名称”。

*现金的价值为现金的金额，实物的价值为该实物的估值

说明奖品分为两种，一种是现金，一种是实物。此处可以用泛化，也可以用关联，如图 9-21。

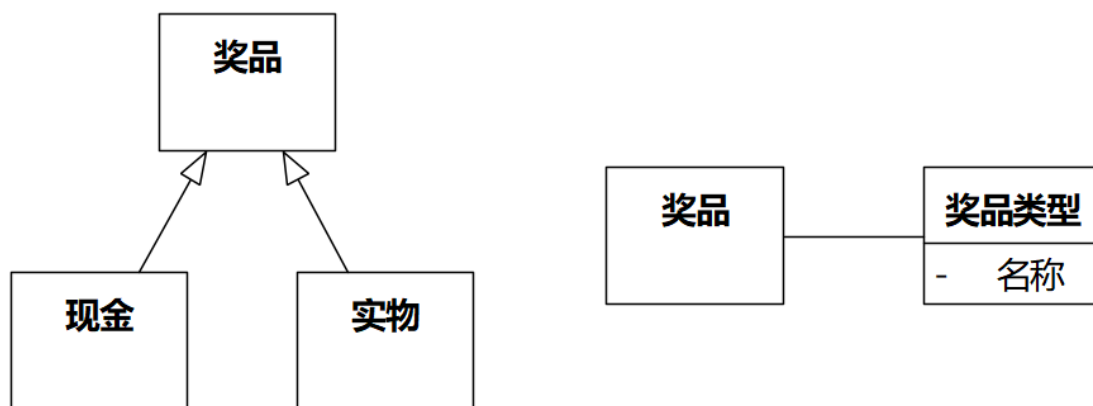


图 9-21 两种奖品

浏览用例规约，判断“奖品”为“现金”还是“实物”不存在系统需要关注的复杂行为差异，因此选择图 9-21 右侧的做法，添加“奖品类型”类，“奖品类型”和“奖品”关联，“奖品类型”的多重性为 1，“奖品”的多重性为*。

*学员成绩

某次“活动”的“学员成绩”可以通过累计“学员”参加该次“活动”的所有“回答”的“得分”计算，不需要提炼新的类或属性。

*活动

提炼为类“活动”。“活动”和“学员”关联，多重性均为*，“活动”和“试卷”关联，多重性均为*。

*中奖次数

可以通过“学员”的“抽奖”统计，不需要提炼新的类或属性。

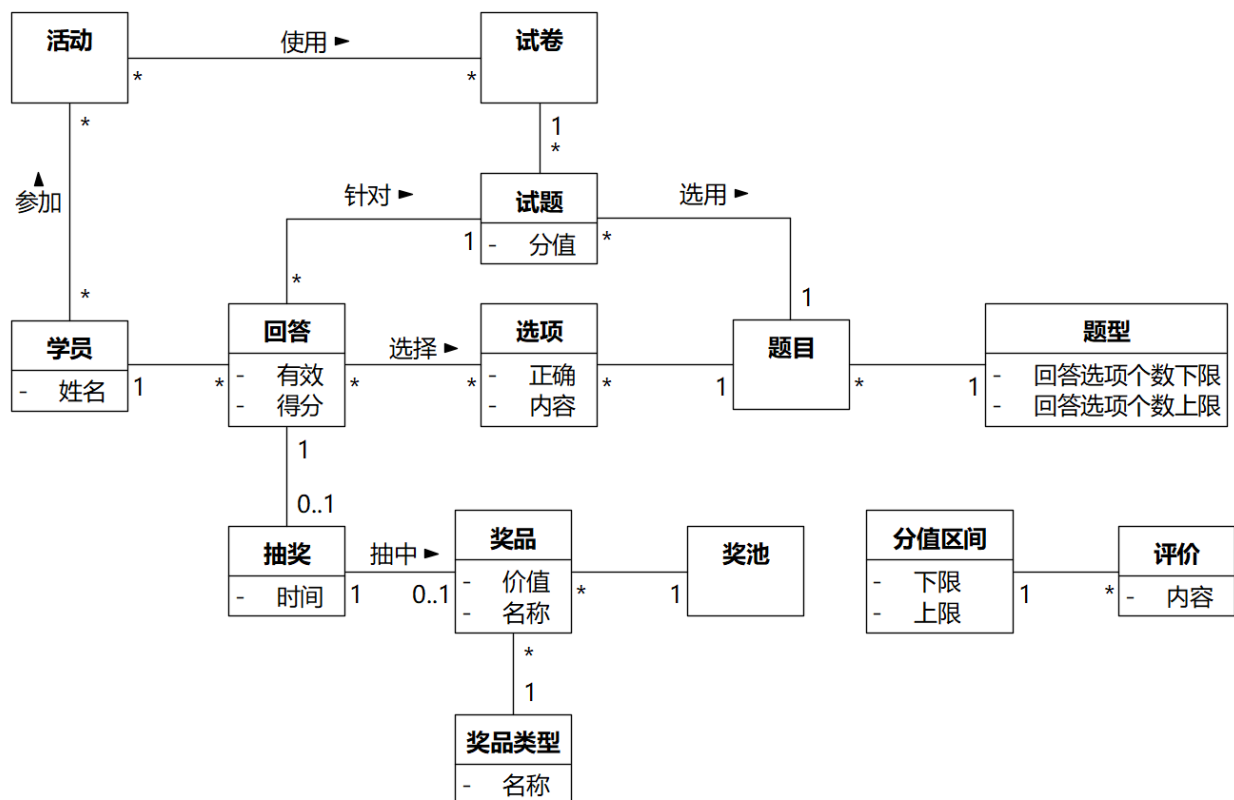


图 9-22 活动

9.1.6.8 步骤 8-11 及其补充约束

用例规约：

8. 系统验证抽中的奖品为现金类型且存在学员的微信号。
9. 系统请求微信向学员的微信号发红包。
10. 系统保存发奖结果。
11. 系统反馈发奖结果。
9. 红包金额为现金奖品的价值。
10. 回答+是否发放成功+发放时间。
11. 是否发放成功。

提炼：

*微信号

提炼为“学员”的属性。

*微信

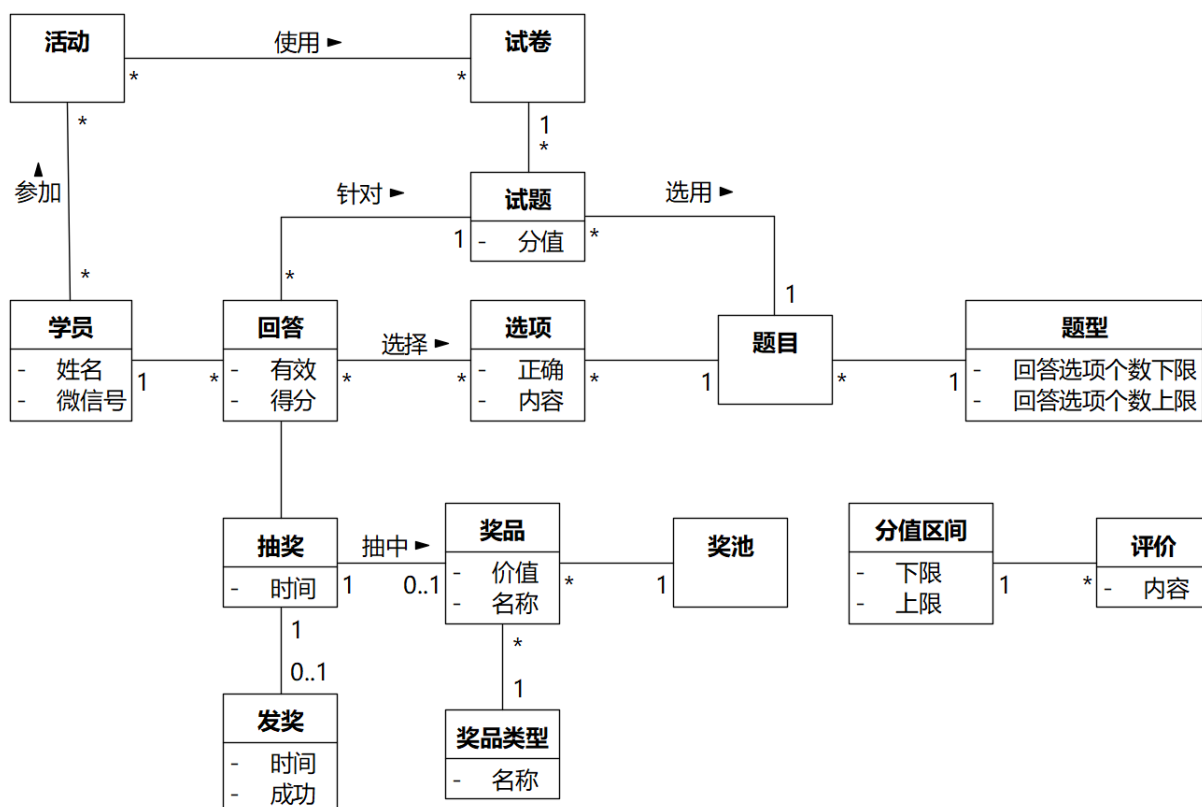
“微信”是一个外系统，提炼出边界类“微信接口”。

*发奖结果

提炼“发奖”类。“发奖”和“抽奖”关联，“发奖”的多重性为0..1，“抽奖”的多重性为1。

*是否发放成功+发放时间

提炼为“发奖”的属性。



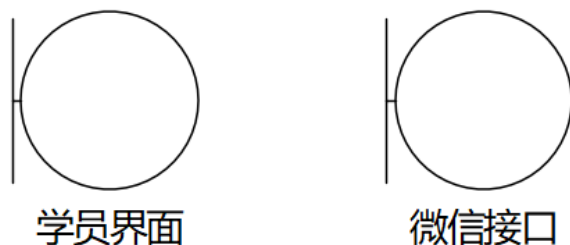


图 9-23 发奖

到此，该用例规约的所有可提炼的内容已提炼完毕。

9.1.6.9 进一步思考

(1) 奖品

系统不需要区分具体的奖品，只需要记住奖品的规格。例如，图 9-24 中的两听可乐，系统不加区分。这里的“奖品”实际上指的是奖品规格，因此需要记住某个“奖品”的数量。



奖品ID	名称	价值
1	百事可乐罐装330ml	3.5
2	红牛250ml	6
3	现金5.12元	5.12
4	乐事原味桶装薯片104g	8
5	脆香米巧克力24g	4.5

奖池项ID	奖品ID	数量
100	1	2
101	2	1
102	3	1

图 9-24 关于奖品的数据示例

那么，“数量”是哪个类的属性呢？

“奖品”能决定“数量”吗？不能。“数量”是“奖品”存在于“奖池”中的“数量”，即（奖品，奖池）→数量。

因此，可以添加一个关联到“奖品”和“奖池”的类“奖池项”，把“数量”作为“奖池项”的属性。“奖池项”和“奖品”关联，“奖池项”的多重性为 0..1（注意，不是*了，因为有了“数量”属性，“奖品”现在是规格），“奖品”的多重性为 1。“奖池项”和“奖池”关联，“奖池项”的多重性为*，“奖池”的多重性为 1。

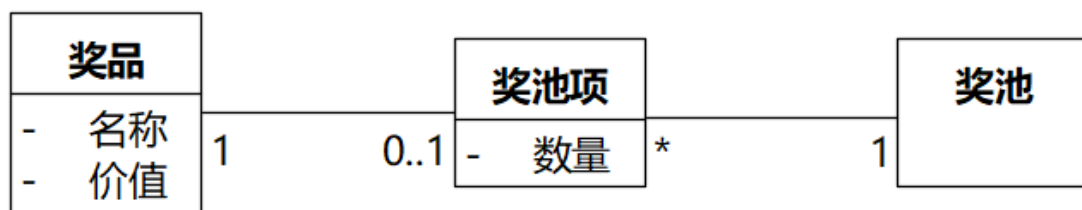


图 9-25 奖池项

本系统只需要一个奖池，可以把“奖池”类删掉，但考虑到需要一个类来承担抽奖的责任，继续保留“奖池”类。

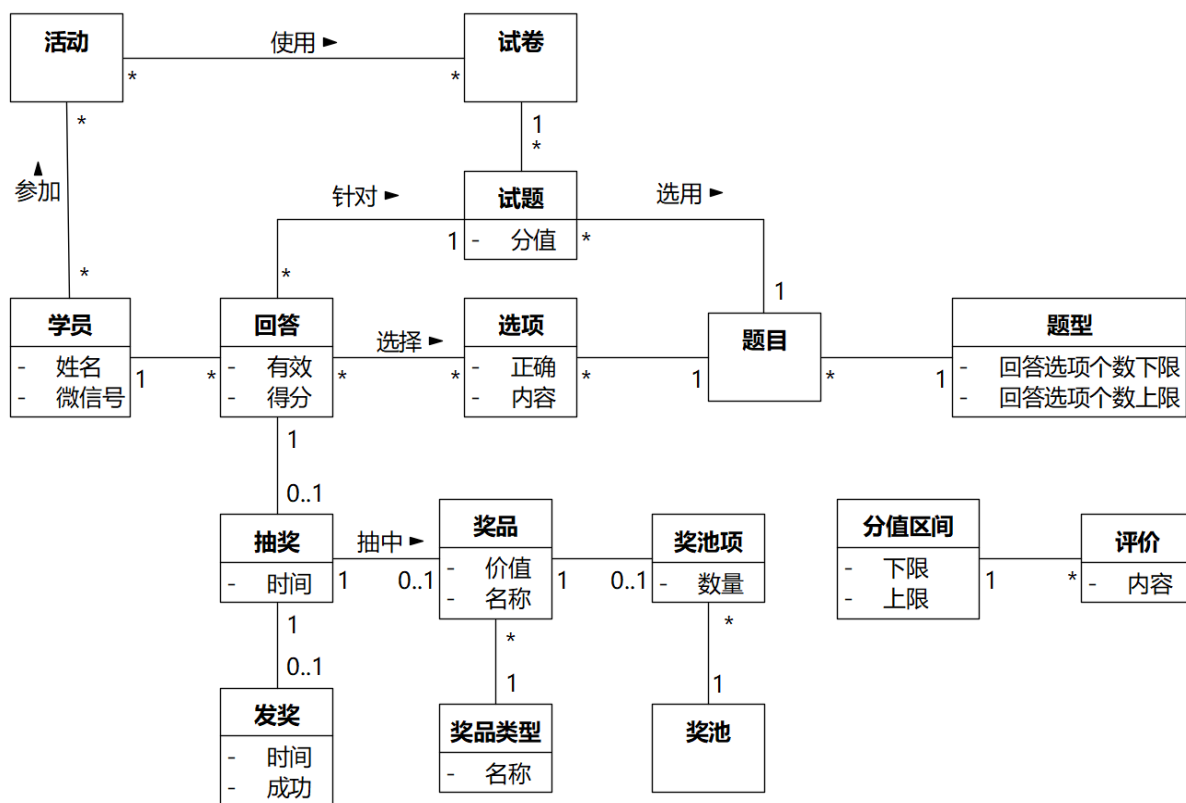


图 9-26 类图进展

(2) 多对多的关联

*学员-活动

学员参加活动的细节不需要考虑，所以此处不需要修改。

*活动-试卷

此处需要修改。学员成绩通过统计某次活动中学员回答问题的得分得到，当前的类图无法推断出某一次回答是在哪一次活动做出的。

例如，一名学员参加了一次“软件需求设计建模方法学全程实例剖析训练”，被老师点名答了一些题，成绩不好，于是，一周后又申请参加了一次“软件需求设计建模方法学全程实例剖析训练”，又答了一些题，这一次成绩大有改善。

这两次“软件需求设计建模方法学全程实例剖析训练”老师使用的试卷是一样的，甚至有可能该学员两次都被点到答同一题，但目前的类图无法区分。

可以有两种做法。第一种如图 9-27，在“活动”和“回答”之间建立关联。

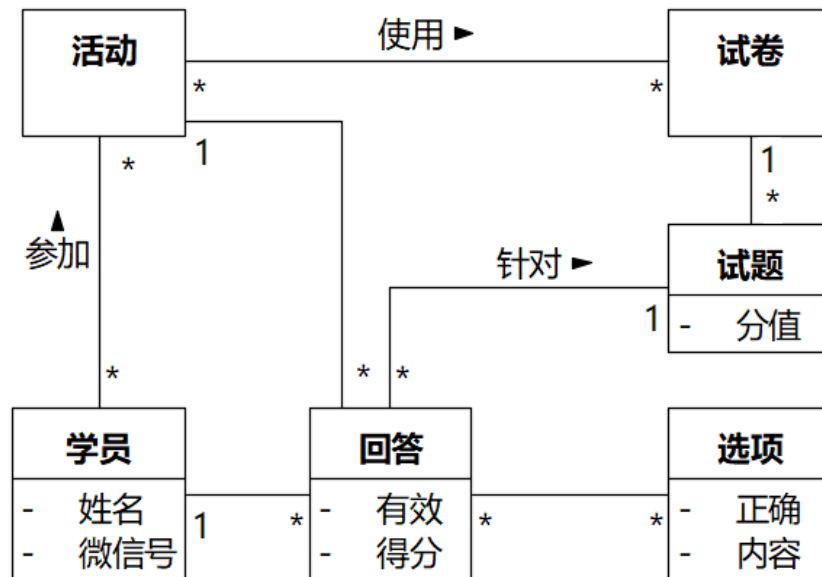


图 9-27 解决活动-试卷问题 之一

第二种如图 9-28，添加一个“考试”类，和“活动”、“试卷”、“回答”关联。

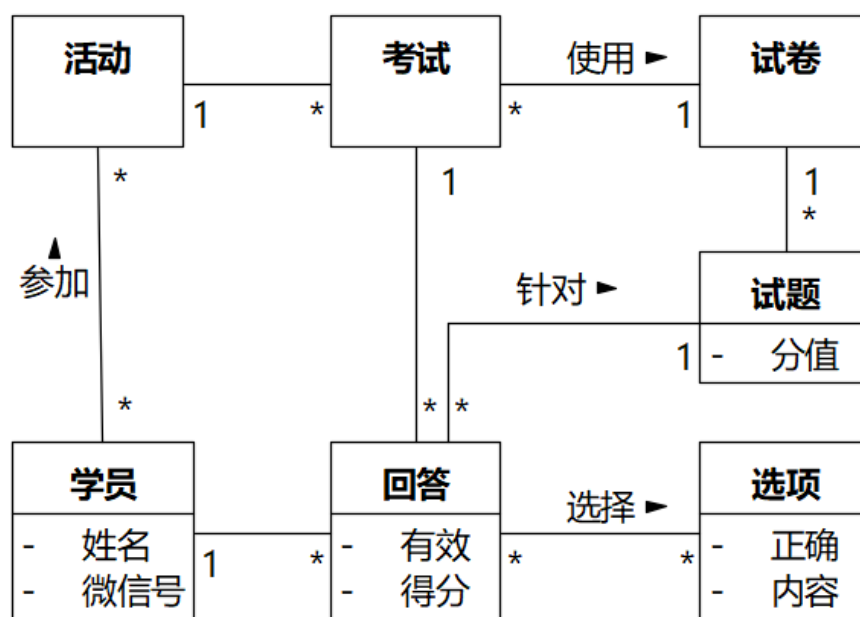


图 9-28 解决活动-试卷问题 之二

我们采用第二种做法，虽然多了一个类，但概念更加清晰。“考试”是在某次“活动”中对某一份“试卷”的使用，产生许多“回答”，同

时，也消除了“活动”和“试卷”之间的多对多关联。

*回答-选项

关联细节不需要考虑，所以此处不需要修改。

目前类图的进展如图 9-29。

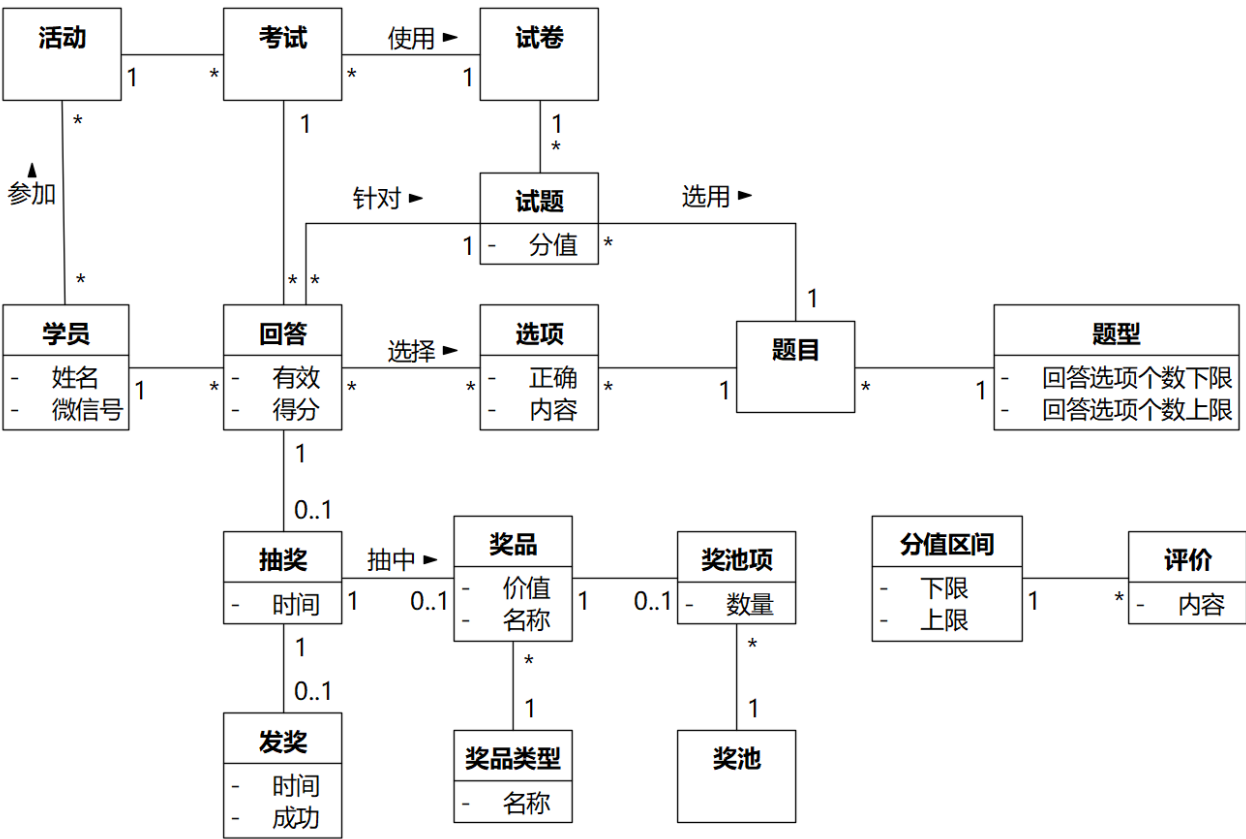


图 9-29 类图进展

(3) 有的类的属性框是空白的

属性框空白不代表没有属性，因为关联也是属性。

可能有人会想，“活动”、“试卷”是不是应该有个“名称”属性，“题目”是不是有个“内容”属性，“考试”要不要加个“时间”属性.....

很可能是有的，但是所给用例的用例规约没有提到，说明该用例没

实际上后面我们会发现，也许根本就不需要聚合/组合的概念。

(6) 如果把系统分为若干个独立的系统，类图上的类该如何分割？

同上，无足够证据，需要继续往下走。