





案例说明

❖ 案例研究

- 餐馆预约管理系统

❖ 包含统一过程定义的4个主要工作流的一次迭代


- 需求 (requirements)
- 分析 (analysis)
- 设计 (design)
- 实现 (implementation)

❖ 通过案例说明UML在软件开发中的使用




餐馆系统

业务建模



Contents

1	Informal Requirements
2	Use Case Modeling
3	Describing Use Cases
4	Domain Modeling
5	Glossaries



业务建模

- ❖ 在开发的早期阶段进行
- ❖ 输入
 - 非正式的规格说明
- ❖ 活动
 - 创建用例模型 (use case model)
 - 定义用例
 - 创建领域模型 (domain model)
 - 创建词汇表 (glossary)



需求（非正式描述）

- ❖ 开发系统的意图
 - 通过改进为顾客预定和分配餐桌的过程，支持某餐馆的日常经营
- ❖ 系统描述
 - 餐馆在晚间供应三次餐点，称为“简餐”、“正餐”和“宵夜”时段；但这些时段无须严格遵守，可以跨时段预约
 - 餐馆现在采用手工预约系统，使用手写预约单，保存在一个大文件夹中
 - 如果有空餐桌，用餐者也可以不提前预约就进餐馆用餐，称为walk-in

预约记录单

DINNER BOOKINGS

DATE TUE 12/3/96

5.30 - 7.30PM			7.45 - 9.45PM			10.00 - 11.30PM		
TIME	COVERS	NAME & PHONE NO.	TIME	COVERS	NAME & PHONE NO.	TIME	COVERS	NAME & PHONE NO.
TABLE 1								
		7.30 x 4 Hit ⁰¹⁸ 9.15 11.00 x 2 Lane 8259367						
TABLE 2								
		8.00 x 2 Hit ⁰¹⁸ 4.47 12.30 x 2 Maria						
TABLE 3								
6.40 x 1	Smith	8.30 x 2 Vine 261 6622	9.30 x 4	Curtis	018 576 1281			
6.30 x 1	WALK-IN	8.30 x 2 Alex ⁰¹⁸ 6.46 10.30	10.00 x 2	Kennedy	018 871 3142			
TABLE 4								
		8.00 x 3 Helen ⁰¹⁸ 6.47 9.12						
TABLE 5								
		7.30 x 2 Graham ⁰¹⁸ 9.15	9.35 x 2	Pinto	018 221 7618			
TABLE 6								
6.40 x 2	WALK-IN	7.30 x 2 WALK-IN	7.30 x 4	Force	018 460 3223			
TABLE 7								
Comments: 10.00								

需求 (非正式描述)

❖ 预约记录单

- 预约单中的每一行对应餐馆中一张餐桌，登记的每个预约对都一张餐桌，一张餐桌在不同时间可多次预约
- 每个预约中记录有餐具的套数(covers)，即预期进餐者的人数，用来分配大小合适的餐桌
- 当用餐者到来并在他们的餐桌就座时，就划掉相应的预约登记。如果他们就座的不是预约的餐桌，就画一个箭头从预约的餐桌指向新餐桌。
- 如果顾客打电话取消预约，并不能从表中真正地擦除，而是在预约上做一个备注“已取消”
- 预约还记录联系人的姓名和电话
- 对walk-in顾客在预约单中仍登记为预约以表示餐桌被占用，但是不记录顾客的姓名或电话



对计算机化系统的需要

❖ 现有手工系统的问题

- 手工系统速度慢，预约登记单很快就变得难以理解
 - 这可能导致经营上的问题，例如，实际上有空餐桌而由于这个预约单不是很明显，会妨碍顾客进行预约。
- 没有备份系统
 - 如果一张预约单被毁坏了，餐馆就没有了当晚有什么预约的记录。
- 从现有的预约单获取即使很简单的管理数据也很费时
 - 例如餐桌的使用率



对计算机化系统的需要

❖ 餐厅希望开发一个现行预约单的自动化版本

- 新系统应该和现有的预约单显示同样的信息，并且格式大致相同，使餐馆员工易于转换到新系统
- 每当记录了新的预约或者对已有预约进行修改时，应该立即更新显示，使餐馆员工在工作时总能获得最新信息
- 系统必须易于记录餐馆营业时发生的有意义的事情，例如顾客的到来
- 系统的操作应当尽可能是直接操作屏幕上显示的数据
 - 例如，可以简单地将预约拖动到屏幕上一个适当的位置来改变预约的时间或者分配的餐桌。



定义第一次迭代

- ❖ 迭代和增量的方法建议：系统的第一次迭代应该只交付足够使系统提供某些确实有商业价值的核心功能。
- ❖ 在餐馆预约系统这个例子中，基本需求是餐馆在营业时记录预约和更新预约单信息
 - 如果这些功能可以使用，就有可能用这个系统代替现有的预约单，然后在后续的迭代中再开发其他功能

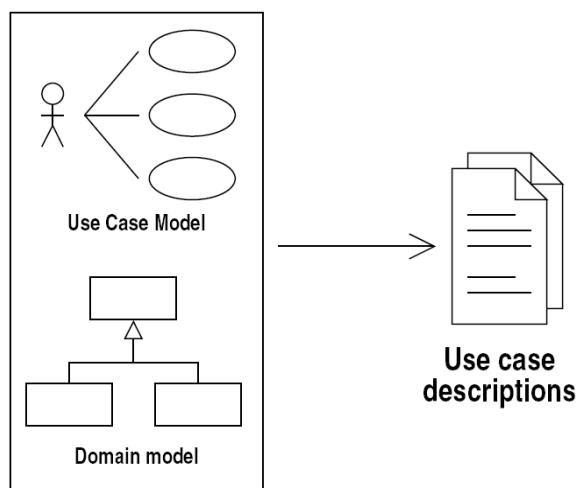


基本功能

- ❖ 记录提前预约
 - 联系人姓名和电话号码
 - 用餐者人数：covers
- ❖ 记录未预约的用餐者walk-in
 - 只记录用餐者人数
- ❖ 为预约分配餐桌
 - 调换餐桌
- ❖ 在预约记录单上备注相关信息
 - 用餐者到达
 - 取消预约等

用例驱动的开发过程

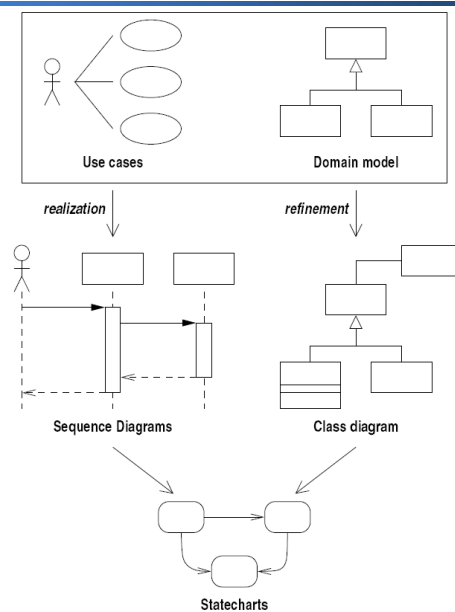
❖ UML捕捉需求




用例驱动的开发过程

❖ 用例实现


❖ 用例细化





用例

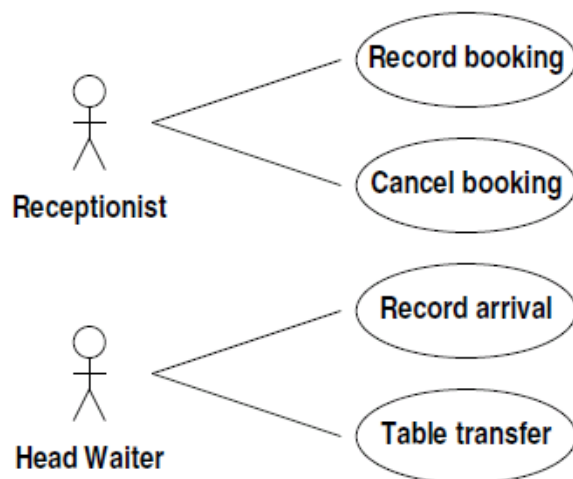
- ❖ 用户与系统交互完成的任务
- ❖ 预约系统（Booking System）的初步用例
 - 记录新预约的信息
 - 取消预约
 - 记录顾客到达
 - 调换顾客的餐桌



参与者

- ❖ 参与者是用户和系统交互时承担的角色
 - Receptionist接线员（进行预约）
 - Head waiter领班（分配餐桌）
- ❖ 注意
 - 一个用户可能在不同时候扮演一个或多个角色
 - 顾客不是这个系统的用户，所以不作为参与者

初步用例图



描述用例

❖ 用例描述

- 用例包括了一个用户在执行给定任务时的所有可能的交互
- 场景 (*scenarios*) 或事件路径 (*courses of events*) 描述
- 以系统和用户之间会话的形式描述

❖ 完整的用例描述包括

- 一个基本 (*basic*) 事件路径：正常情况下的场景
- 多个可选 (*alternative*) 事件路径：分支和可预料的其他情景
- 多个异常 (*exceptional*) 事件路径：出错情况



记录预约：基本事件路径

❖ Record booking: basic course of events

1. *The receptionist enters the date of the requested reservation.*
2. *The system displays the bookings for that date.*
3. *There is a suitable table available; the receptionist enters the customer's name and phone number, the time of the booking, the number of covers and the table number.*
4. *The system records and displays the new booking.*



记录预约：可选事件路径

❖ Record booking - no table available: alternative course of events

1. *The receptionist enters the date of the requested reservation.*
2. *The system displays the bookings for that date.*
3. *No suitable table is available, and the use case terminates.*



记录预约：异常事件路径

❖ Record booking – table too small: exceptional course of events

1. The receptionist enters the date of the requested reservation.
2. The system displays the bookings for that date.
3. The receptionist enters the customer's name and phone number, the time of the booking, the number of covers and the table number.
4. The number of covers entered for the booking is larger than the maximum specified size of the table required, so the system issues a warning message asking the user if they want to continue with the reservation.
5. If the answer is 'no', the use case will terminate with no booking being made.
6. If the response is 'yes', the booking will be entered with a warning flag attached.



UI原型

- ❖ 事件路径描述了用户和系统之间的交互，但是没有明确地详述这些交互是如何发生的
- ❖ 一般而言，在用例描述中详述用户界面不是个好主意
 - 用例描述的重点是定义系统和用户之间交互的总体结构，如果包含用户界面的细节会使之不清晰
 - 用户界面设计应该协调一致便于使用，这只有在合理地考虑了各种用户任务后才能做到
 - 如果用例描述不适当地指定了用户界面的细节，可能会使用户界面设计者的工作更加困难，或者需要大量改写用例描述

UI原型

❖ 不过，对用户界面是什么样的有个大概看法，可能会有助于理解用例描述

Booking System													
Booking	Date: 10 Feb 2004												
	18	:30	19	:30	20	:30	21	:30	22	:30	23	:30	24
1													
2			Ms Blue 0121 7648 4495										
			Covers: 3										
3							Mr White 0865 364795						
							Covers: 2						
4			Mr Black 020 8453 7646										
			Covers: 4										
5			Walk-in										
							Covers: 2						


记录到达：基本事件路径

❖ **Record arrival: basic course of events**

1. The head waiter enters the current date.
2. The system displays the bookings for that date.
3. The head waiter confirms arrival for a selected booking.
4. The system records this and updates the display, marking the customer as having arrived.

❖ 和“记录预约”用例的前两个步骤相同

- 参与者不同



用例包含

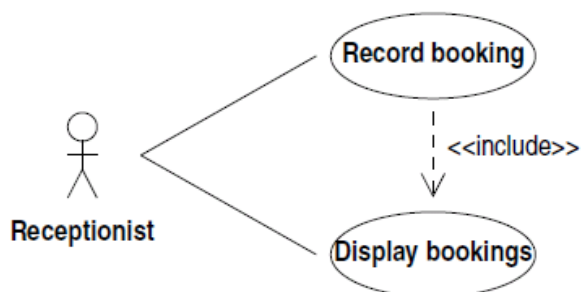
- ❖ 用例的公共行为
 - 如果多个用例中的公共交互本身能形成一个完整用例，可以使用用例包含
- ❖ 将“记录预约”和“记录到达”两个用例中共享的步骤抽出，形成用例“显示预约”
- ❖ **Display bookings: basic course of events**
 1. The user enters a date.
 2. The system displays the bookings for that date.



修改用例描述

- ❖ 在用例描述中包含其他用例
- ❖ **Record booking: basic course of events (revised)**
 1. The receptionist *performs the 'Display bookings' use case.*
 2. The receptionist enters the customer name and phone number, the time of the booking, the number of covers and the table reserved.
 3. The system records and displays the new booking.

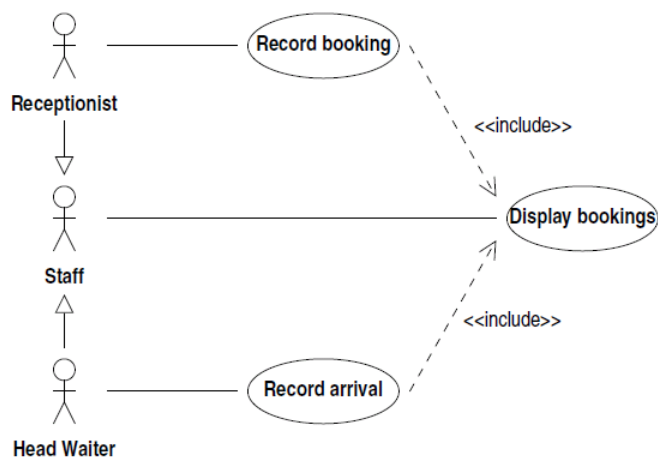
修改用例图



参与者泛化


- ❖ 图中显示接线员可以执行“显示预约”，同样，领班也可以显示预约
- ❖ 引入一个更一般的参与者，表示这两个参与者的公共行为特征

修改用例图和用例描述




用例扩展

- ❖ “记录到达”用例的变体
 - 没有预约的顾客来用餐，有空闲餐桌
- ❖ 用可选事件路径描述
 - 记录到达——没有提前预定：可选事件路径
 1. 侍者领班输入当前日期；
 2. 系统显示当天的预约；
 3. 系统中没有记录该顾客的预约，所以侍者领班输入预约时间、用餐人数和餐桌号，创建一个未预约登记；
 4. 系统记录并显示新预约。
- ❖ 也可以用单独一个用例来描述



用例扩展

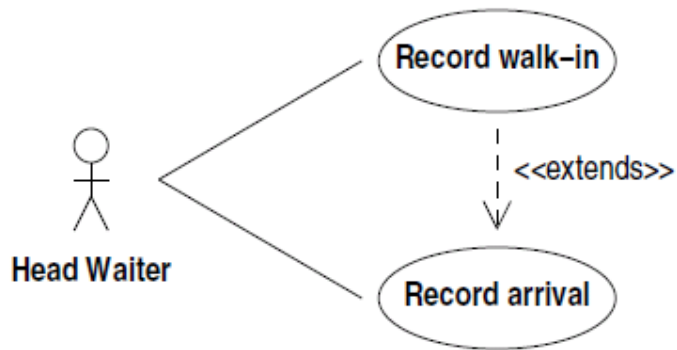
- ❖ Record walk-in: basic course of events
 1. The head waiter performs the 'Display bookings' use case.
 2. The head waiter enters the time, the number of covers and the table allocated to the customer.
 3. The system records and displays the new booking.
- ❖ “记录到达”和“记录未预约顾客”两个用例之间是什么关系呢？
 - include?



用例扩展

- ❖ 包含依赖对这种情况并不适合
 - 因为在“记录未预约顾客”中指定的交互不是在每次执行“记录到达”时都执行
- ❖ “记录未预约顾客”和“记录到达”之间是一种可选的关系
 - “记录未预约顾客”用例只是在“记录到达”的某些情况下被执行：
 1. 对该顾客没有已经记录的预约
 2. 有一张合适的空闲餐桌
 3. 顾客想在餐馆用餐

用例扩展



取消预约用例

❖ Cancel booking: basic course of events

1. The receptionist selects the required booking.
2. The receptionist cancels the booking.
3. The system asks the receptionist to confirm the cancellation.
4. The receptionist answers 'yes', so the system records the cancellation and updates the display.




取消预约用例

- ❖ **Cancel bookings - invalid date: exceptional course of events**
 1. *The receptionist performs the Display Bookings use case for the date given by the customer.*
 2. *The receptionist selects and cancels the relevant reservation.*
 3. *The date of the booking is prior to the current date: the system displays a suitable error message and the booking is not cancelled.*




取消预约用例

- ❖ **Cancel bookings - booking arrived: exceptional course of events**
 1. *The receptionist performs the Display Bookings use case for the date given by the customer.*
 2. *The receptionist selects and cancels the relevant reservation.*
 3. *The booking is already recorded as having arrived: the system displays a suitable error message and the booking is not cancelled*




调换餐桌用例

- ❖ **Table transfer: basic course of events**
 1. The head waiter selects the required booking.
 2. The head waiter changes the table allocation of the booking.
 3. The system records the alteration and updates the display.
- ❖ 根据餐馆的业务规则可以给出其可选和异常事件路径




调换餐桌用例

- ❖ **Table transfer - table too small: exceptional course of events**
 1. The head waiter performs the Display Bookings use case for the current date.
 2. The head waiter selects the required booking.
 3. The head waiter changes the table allocation of the booking.
 4. The number of covers recorded for the booking is larger than the maximum specified size of the new table, so the system issues a warning message asking the head waiter if the transfer should go ahead.
 5. If the answer is `no', the booking is not modified.
 6. If the answer is `yes', the system records the alteration and updates the display.



调换餐桌用例

- ❖ **Table transfer - invalid date: exceptional course of events**
 1. The head waiter performs the Display Bookings use case for the current date.
 2. The head waiter selects the required booking.
 3. The head waiter changes the table allocation of the booking.
 4. The date of the booking is prior to the current date: the system displays a suitable error message and the booking is not modified.



调换餐桌用例

- ❖ **Table transfer - double booked table: exceptional course of events**
 1. The head waiter performs the Display Bookings use case for the current date.
 2. The head waiter selects the required booking.
 3. The head waiter changes the table allocation of the booking.
 4. The head waiter has attempted to transfer the booking to a table that is already occupied: the system displays a suitable error message and the booking is not modified.

完成用例模型

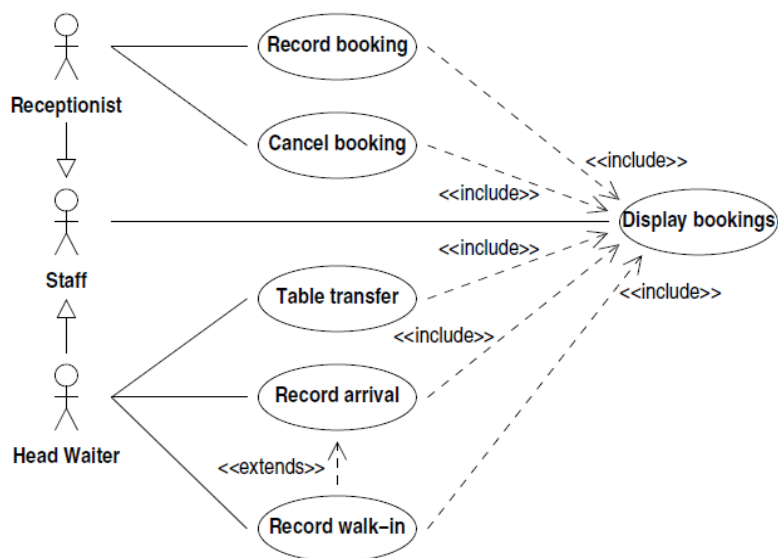
❖ 用例模型

- 用例图
- 用例描述

❖ 什么时候完成用例建模？

- 用例分析是一项非形式化的技术，在一定时间之后再花时间寻求对模型的改进时会降低回报
- 这对包含关系和扩展关系尤其适用：这些关系通常与从用例产生的设计的结构特性并不相当，所以缺少一个可能的依赖的后果并不严重

最后的用例图





用例建模小结

❖ 用例视图

- 捕捉系统、子系统、类或构件对外部用户显现的行为
- 用例视图将系统功能划分为对参与者有意义的交互


❖ 4+1视图

❖ 用例驱动的开发




用例建模小结……参与者

- ❖ 参与者定义了一组在功能上密切相关的角色，当某实体与系统交互时，该实体扮演这样的角色。
- ❖ 参与者可以包括人、外部系统和设备。
- ❖ 参与者是由角色定义的一种类型，它有实例，即承担参与者角色的具体事物。
- ❖ 参与者虽然在用例模型中使用，但参与者并不是系统的一部分，它们位于系统之外，是在系统外部与系统进行交互的事物。



用例建模小结……识别参与者

- ❖ 人员
 - 在直接使用系统的人员发现参与者
 - 例如启动、关闭和维护系统，从系统中获得信息，向系统提供信息
 - 首先关注启动系统的参与者，从中找出后续与系统交互的其他参与者
 - 从用户的角度考虑如何使用系统，特别考虑其中可能发现的三种参与者



用例建模小结……识别参与者

- ❖ 外部系统
 - 所有与本系统交互的外部系统都是参与者
 - 外部系统可以是相对于正在开发的系统的其他子系统、上级系统或下级系统，即任何与当前系统协作的系统。
- ❖ 设备
 - 与系统交互的设备：与系统相连，向系统提供外界信息或在系统的控制下运行，这样的设备是系统的参与者
 - 由操作系统管理的一般标准用户界面设备不包括在内



用例建模小结……用例的特性

- ❖ 用例是描述系统的一项功能的一组动作序列，动作序列表示参与者与系统之间的交互，系统执行该动作序列要为参与者产生结果。
- ❖ 用例是一种类型，它是要实例化执行的。
 - 当参与者实例与系统进行交互时，一个用例描述的功能的全部或部分才发挥作用，其中经历的动作序列即该用例的一个实例。



用例建模小结……用例的特性

- ❖ 用例名是以动宾短语形式出现的。
- ❖ 用例描述的是参与者所使用的一项系统功能，该功能应该相对完整。
 - 用例是一项功能的完整说明，而不能只是其中的一个片段。
 - 不能像结构化分析方法中把大的加工细分为下层的较小加工那样来细分用例。
 - 用例是不分层的，不能说上层的用例由下层的较小用例组成。



用例建模小结……用例的特性

- ❖ 用例只描述参与者和系统彼此为对方直接做了什么事情，不描述怎么做，也不描述间接地做了些什么。
- ❖ 不存在没有参与者的用例，用例不应该自动启动，也不应该主动启动另一个用例。
- ❖ 用例的执行结果对参与者来说是可观测的和有意义的。
- ❖ 用例对参与者产生结果，是指系统对参与者的动作要做出响应。
- ❖ 用例是相对独立的。
- ❖ 一个用例就是一个需求单元、分析单元、设计单元、开发单元、测试单元，甚至部署单元。



用例建模小结……用例之间的关系

- ❖ 用例之间可以存在三种关系
 - 包含、扩展和泛化
- ❖ 泛化
 - 用例之间的泛化关系和类之间或参与者之间的泛化关系一样
 - 特殊用例（子用例）继承了一般用例（父用例）的行为，还可以增加行为或覆盖父用例的行为。



用例建模小结……用例之间的关系

❖ 包含


- 用例中经常存在着重复的动作序列，为了避免重复，把重复的动作序列放在单独一个用例中，原有的用例（基用例）再引入该用例（供应者用例、被包含用例）。
- 一个用例可以包含多个用例，一个用例也可以被多个用例包含。
- 包含关系表明：基用例在它内部说明的某一个或某一些位置上，要使用供应者用例执行的结果。



用例建模小结……用例之间的关系

❖ 扩展

- 在一个或几个用例的描述中，有时存在着可选的描述系统交互行为的动作序列片段。在这种情况下，可以从用例中把可选的动作序列片段抽取出来，放在另一个用例（扩展用例）中，原来的用例（基用例）再用其进行扩展
- 扩展关系表明：按基用例中指定的扩展条件，把扩展用例的动作序列插入到基用例中的扩展点处。
- 扩展点是用例中的一个位置，在这样的位置上，如果该处的扩展条件为真，就要插入扩展用例中描述的全部动作序列或其中的一部分，并执行。执行完成后，基用例继续执行扩展点下面的行为。如果扩展条件为假，扩展不会发生



领域建模

- ❖ 用UML构造现实世界的模型
 - 与实体关系建模类似
- ❖ 建模的结果是类图
- ❖ 无缝开发 (seamless development)
 - 分析和设计使用相同的符号和表示法
 - 设计可以从最初的领域模型演化得到

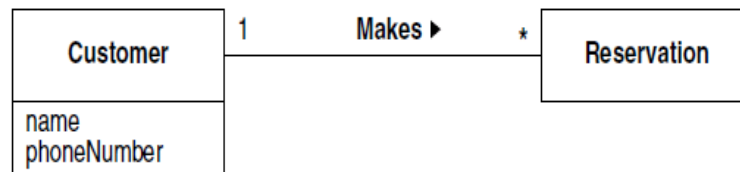
领域模型表示

❖ 使用类图表示法的子集

- **classes** : 表示现实世界中的实体
- **associations** : 表示实体之间的关系
- **attributes** : 表示实体保存的数据
- **generalization** : 用于简化模型的结构

Customers and Reservations

❖ 基本业务事实: customers make reservations



定义关系

- ❖ 为关系命名
 - 使用动词，可以将关系读作一个句子
- ❖ 一名顾客可以有几次预约
 - A customer can make many reservations
 - 用重数指定
- ❖ 一个预约有多少人
 - How many people make a reservation?
 - 可以将预期的用餐人数作为reservation的属性建模

Tables

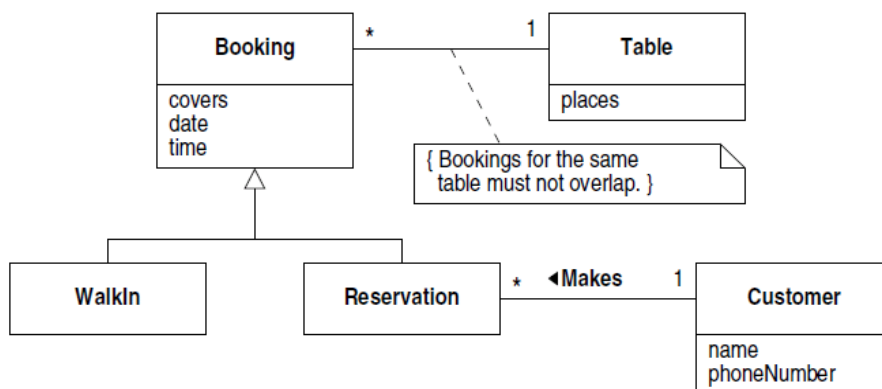
- ❖ 餐桌号是Reservation的属性吗?
- ❖ 餐桌最好是作为单独一个类建模
 - 即使没有预约餐桌也存在
 - 可以保存餐桌的其他属性，例如 size

```

classDiagram
    class Customer {
        name
        phoneNumber
    }
    class Reservation {
        covers
        date
        time
    }
    class Table {
        places
    }
    Customer "1" -- "*" Reservation : Makes
    Reservation "*" -- "1" Table
    note for Reservation, Table "Reservations for the same table must not overlap."
  
```


泛化的使用

- ❖ 可以用超类说明不同类型的Booking共享的特性




正确性

- ❖ 我们如何得知领域模型什么时候完成？
 - 无法知道：大多数情况下可能的模型有很多种
- ❖ 领域建模本身并不是终点，而是后续开发的指导
- ❖ 实现用例可以测试领域模型，通常导致进一步的精化



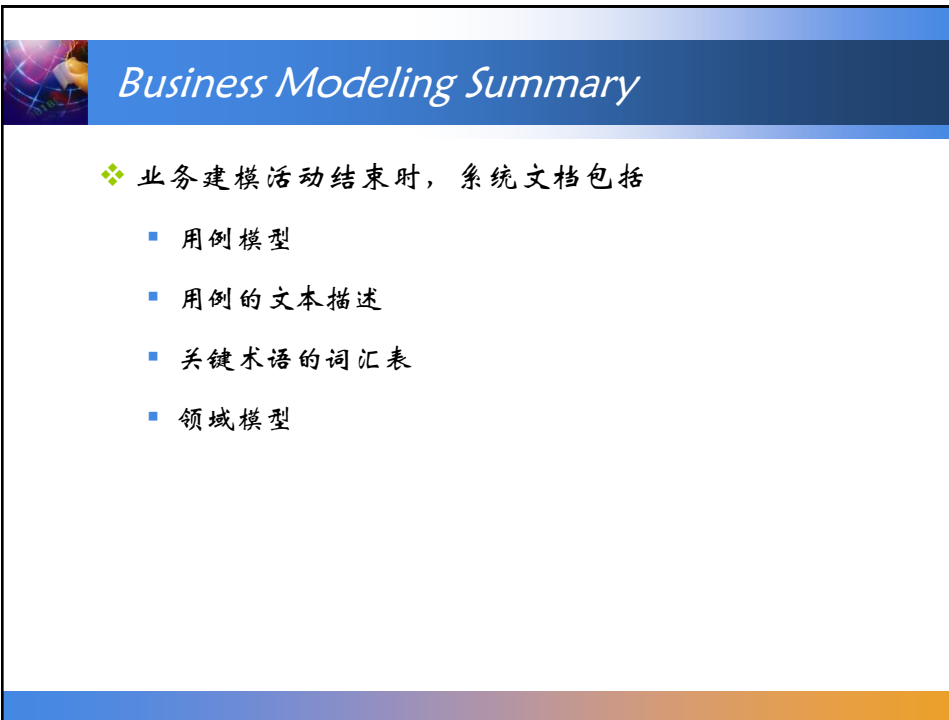
词汇表

- ❖ 领域模型捕捉重要的系统概念
- ❖ 在整个项目开发过程中，领域模型对于记录重要术语和它们的定义很有用
- ❖ 可以用词汇表（*glossary*）的形式表示



Restaurant Glossary

- ❖ **Booking** An assignment of a table to a party of diners for a meal.
- ❖ **Covers** The number of diners that a reservation is made for.
- ❖ **Customer** The person making a reservation.
- ❖ **Diner** A person eating at the restaurant.
- ❖ **Places** The number of diners that can be seated at a particular table.
- ❖ **Reservation** An advance booking for a table at a particular time.
- ❖ **Walk-in** A booking that is not made in advance.



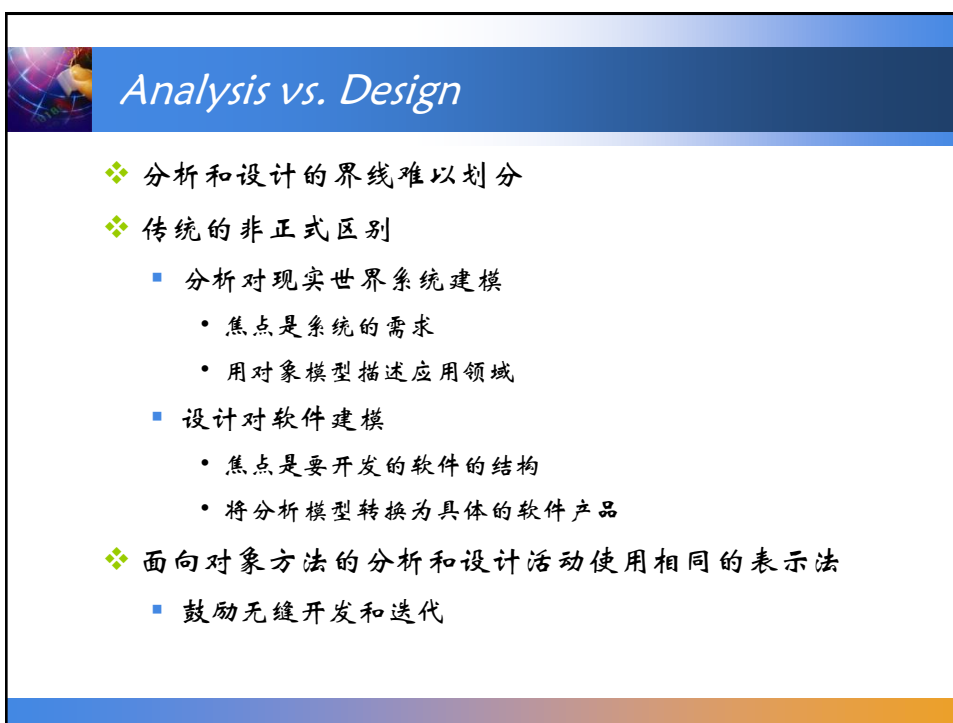
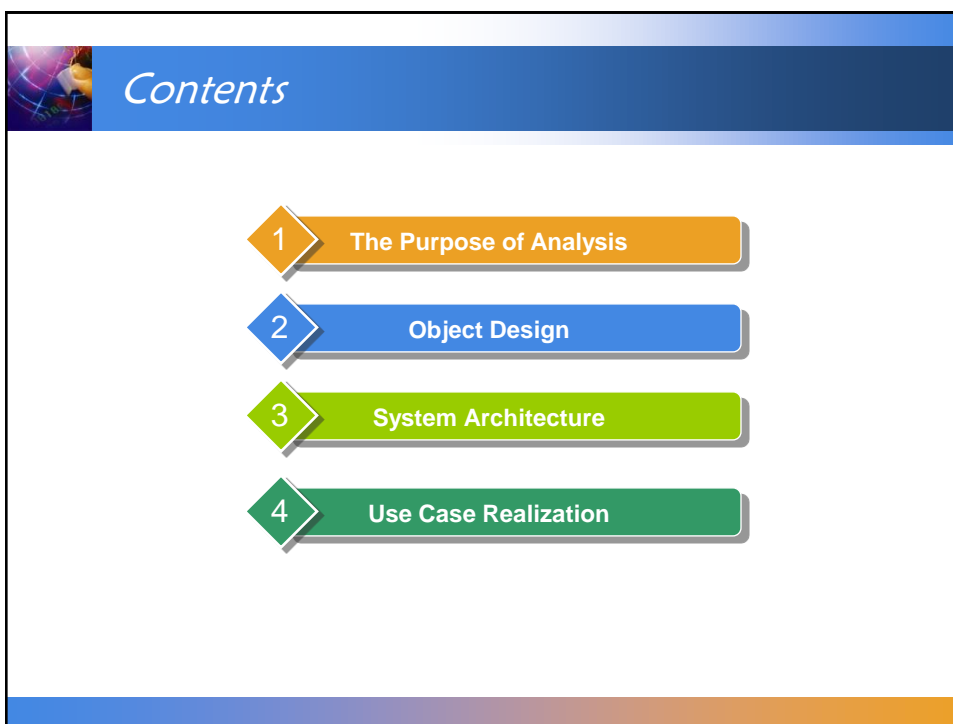
Business Modeling Summary


- ❖ 业务建模活动结束后，系统文档包括
 - 用例模型
 - 用例的文本描述
 - 关键术语的词汇表
 - 领域模型



餐馆系统


分析





分析

- ❖ 要分析什么？ (What)
 - 系统的需求
- ❖ 为什么？ (Why)
 - 论证需求的可实现性
- ❖ 怎么分析？ (How)
 - 绘制交互图，实现用例
 - 实现用例时，为每个用例开发高层交互，论证所需的系统行为如何通过适当的类的实例的交互而产生



分析的目的

- ❖ 分析活动的任务
 - 构造模型，以此论证相互作用的对象如何产生用例指定的行为的任务
- ❖ 分析活动的典型输入是用例和领域模型
 - 用例描述呈现从外部看到的系统的功能，以用户和系统交互的方式表现（行为）
 - 领域模型定义了重要业务概念之间的关系（结构）
 - 缺少的是一个详细说明：表示或源自业务实体的对象如何协作并以这种方式实现用例指定的行为？




分析的目的

- ❖ 分析最重要的任务
 - 产生用例实现
 - 将领域模型演化为更详尽的类图
- ❖ 统一过程的分析工作流程中还包括产生系统的架构描述的 (*architectural description*)
 - 架构描述是对系统总体结构的高层次的决策
 - 不是个别用例会被如何处理的详细说明




对象设计

- ❖ 对模型中的每个类要定义属性和操作
 - 为了产生实现用例的交互图，所需要的数据和处理必须分配到一组对象中，这组对象再相互作用支持用例规定的功能
- ❖ 可以从领域模型开始，但是
 - 现实世界应用的结构并不总是软件系统的最优化结构
 - 领域模型中没有说明操作
- ❖ 用例实现能发现和识别操作并确认设计能支持需要的功能



对象的职责

- ❖ 系统中的每个类都应该有良定义的职责
 - 管理系统数据的一个子集
 - 属性值，到其他对象的链接
 - 管理某些处理
 - 计算、协调
- ❖ 类的职责应该是内聚的 (*cohesive*)
 - 应该 “belong together”
 - 应该形成一个有意义的整体



软件架构

- ❖ 统一过程的分析工作流程包括产生架构描述
- ❖ 架构描述定义
 - 子系统的顶层结构
 - 这些子系统的角色和相互作用
- ❖ 典型的架构可以用模式 (*pattern*) 系统化地整理描述
 - 例如，分层架构 (*layered architectures*)

分层架构

❖ 为什么定义层次

- 将职责分配到不同的子系统中
- 定义层次之间的清晰、简单的接口

❖ MVC 架构

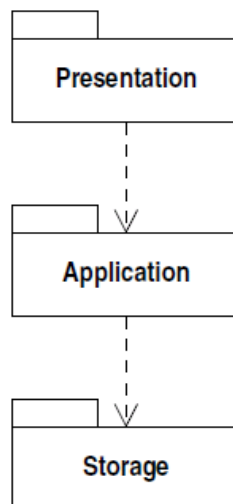
- 在系统的各组成部件之间做出了清晰的区分，有的负责维护数据，有的负责将数据呈现给用户
 - *model class*，负责维护数据
 - *view class*，负责显示数据
 - *controller class*，负责控制复杂的交互


分层架构

❖ 子系统用UML包和依赖表示

❖ 没有构造型的依赖意思是 *uses*


- 较高层可以利用较低层提供的特征
- 较低层的定义独立于较高层





分隔关注

- ❖ 分层的目的是让系统不受变更的影响
- ❖ MVC的关键思想
 - 模型model应该独立于视图view
 - 例如，用户界面经常变化，而应用层不使用表示层，所以对系统的修改可以限制在应用层的类中
 - 类似地，持久数据存储层也是从应用逻辑中分离出来的



分隔关注

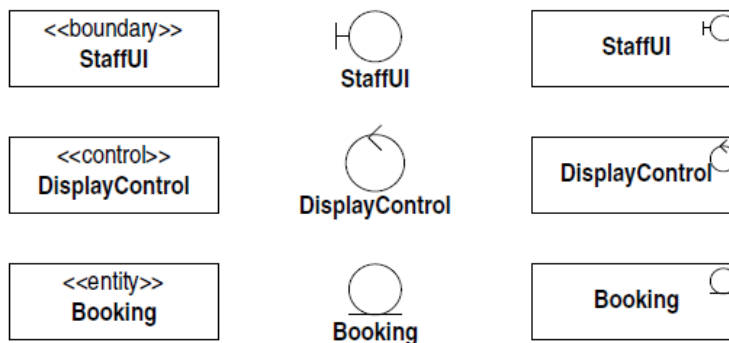
- ❖ 在三层架构中分析和设计的区别
 - 分析一般只关注应用层的对象的行为和交互
 - 对每个系统一般是独特的
 - 分析是一种论证所提议的系统实际上可行的方法
 - 设计关注所有层上的对象设计，还特别关注层与层之间的交互
 - 表示层和存储层的需求对许多系统都是共通的
 - 与分析相比，在很大程度上，可以应用以前做过的工作产生设计

分析类构造型

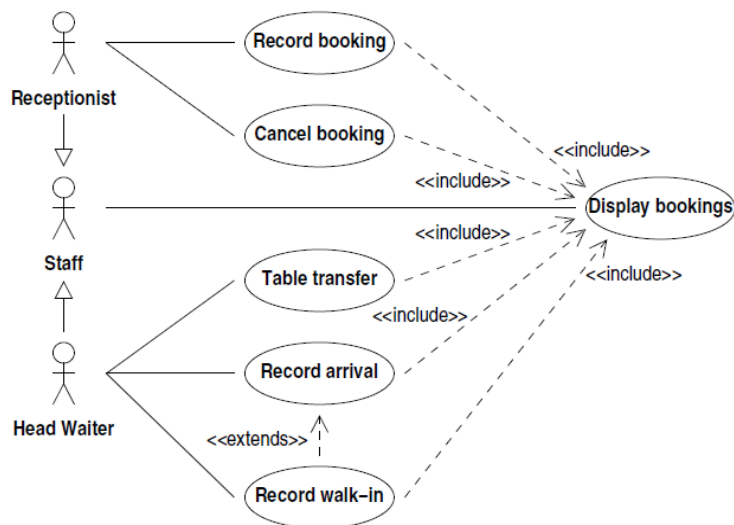
- ❖ 在这个架构中，对象有多种角色
 - 边界对象 (*boundary objects*) 和外部参与者交互
 - 是UI的抽象，处理输入和输出
 - 控制对象 (*control objects*) 管理用例的行为
 - 控制应用层上的一个用例中涉及的交互
 - 实体对象 (*entity objects*) 维护数据
- ❖ 在UML中对象的角色用分析类构造型明确表示

分析构造型的表示法

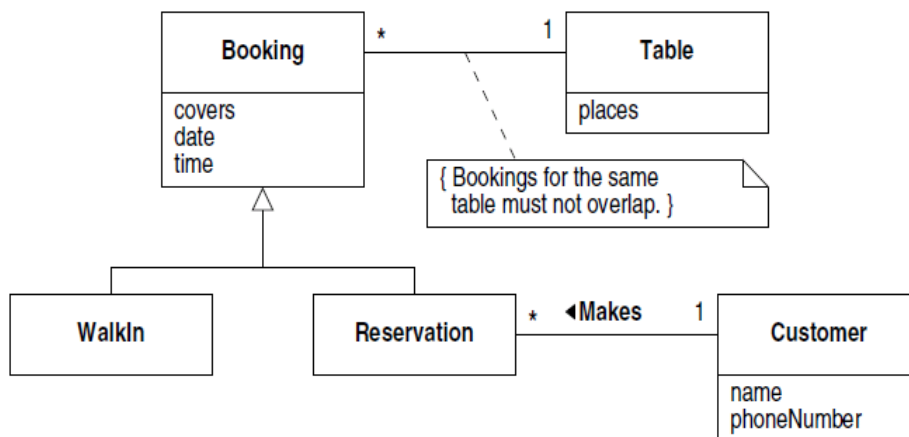
- ❖ 构造型可以是文本或图标符号
- ❖ 图标符号可以代替正常的类方框



Use Case Diagram

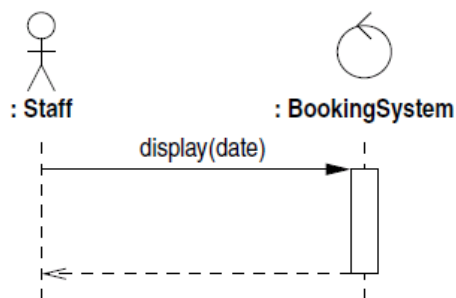


领域模型



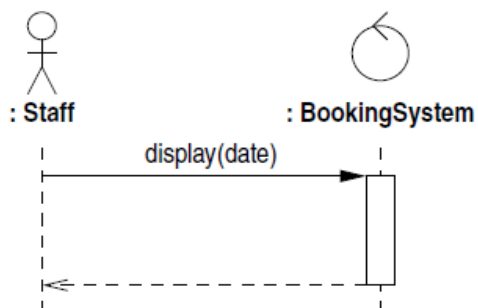
用例实现

- ❖ 从应用层中的功能开始
- ❖ ‘Display Bookings’ 的简单对话
 - 用户提供需要的数据
 - 系统响应是更新显示
- ❖ 最初的实现包括
 - ‘Staff’ actor的实例
 - 一个表示系统的对象
 - 它们之间传递的消息



系统消息

- ❖ 系统消息
 - 从参与者发送到系统的消息
- ❖ 系统用一个controller表示
 - 开始时分析用例行为，而不是输入/输出

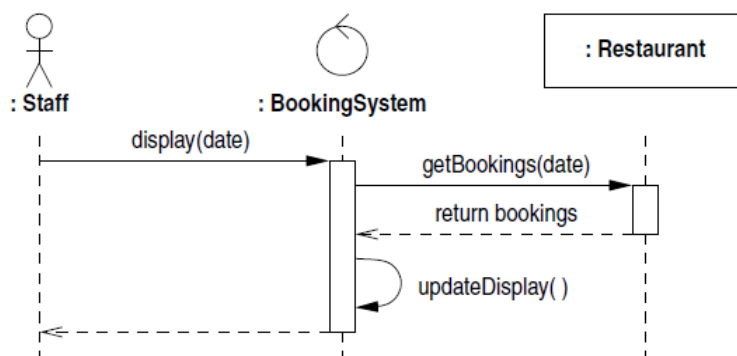


访问预约

- ❖ 系统如何检索预约并显示？
- ❖ 哪个对象应该有责任保存和记录所有的预约？
 - 如果将保存和记录所有预约作为 ‘BookingSystem’ 对象的另一项职责，那么会有损内聚度
 - 所以定义一个新的 ‘Restaurant’ 对象来负责管理预约数据

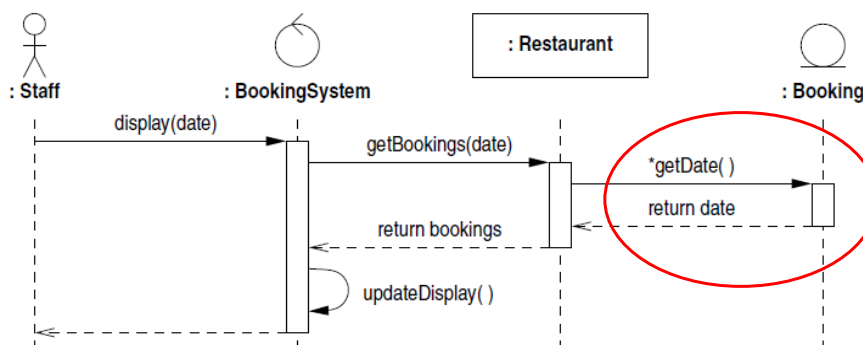
检索预约

- ❖ 增加一条消息来获取相关的预约
- ❖ ‘updateDisplay’ 是内部消息
 - 由系统中的对象产生



检索预约详细信息

❖ ‘Restaurant’ 对象需要检查预约的日期



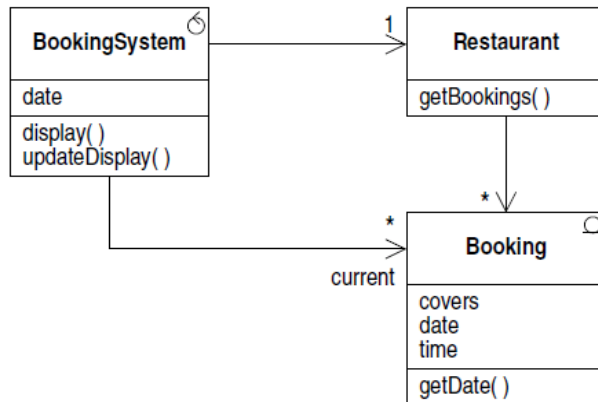
细化领域模型

❖ 上面的用例实现中包括

- 新的'Restaurant'类和'BookingSystem'类,二者之间有一个关联
- ‘Restaurant’类到‘Booking’类的一个关联
 - ‘Restaurant’维护到所有预约的链接
 - 从restaurant向booking发消息
- ‘BookingSystem’到‘Booking’的一个关联
 - ‘BookingSystem’维护到当前显示的预约的链接

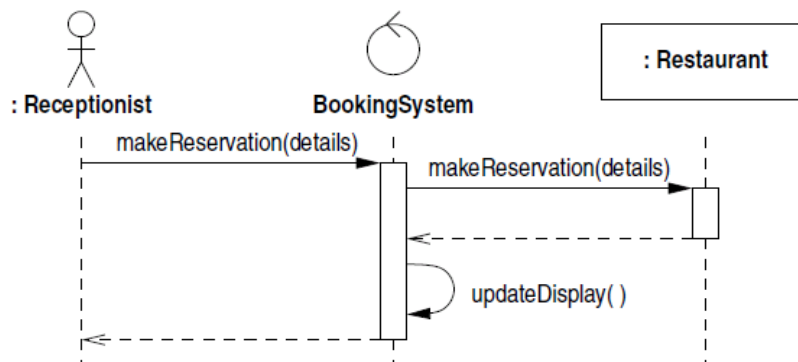
更新类图

- ### ❖ 可以从发送给类实例的消息导出操作



记录新预约

- ❖ 由 ‘Restaurant’ 负责创建新预约
 - 对用户输入或数据的细节先不建模

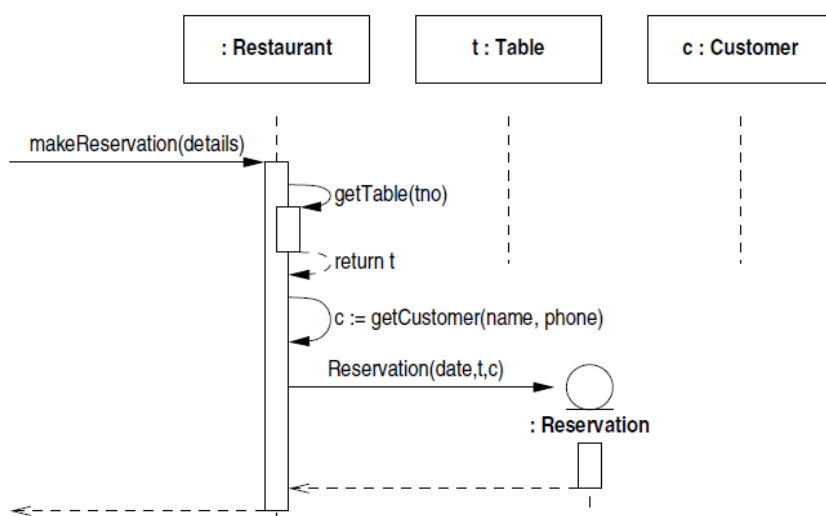


创建新预约

❖ 预约必须链接到餐桌和顾客对象

- 应该由哪个对象负责提供对餐桌和顾客对象的存取？
- 如果给定了预约详情的识别数据，那么由
 - ‘Restaurant’ 负责检索这些数据
 - 潜在的危险：Restaurant对象最后可能有内聚性相当低的责任集合
 - 此时似乎没有理由将这些责任分给多个不同的类

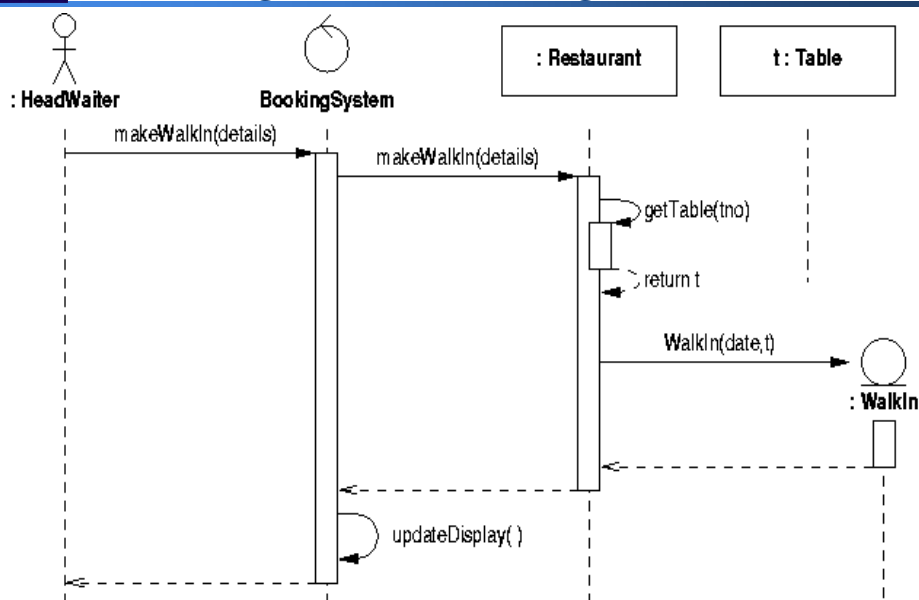
创建新预约



记录 Walk-in 预约

- ❖ Walk-in bookings 是在顾客没有提前预约就到餐馆用餐时创建的
 - 在系统中和 reservation 一样记录
 - 没有与 customer 的关联，创建时需要的信息少一些

Recording Walk-in Bookings



取消预约

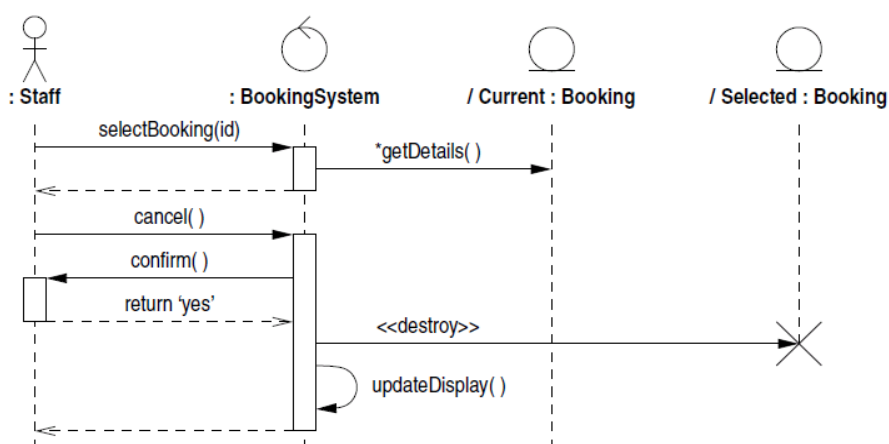
❖ 取消过程有三个步骤

1. 在屏幕上选择要取消的预约
2. 要求用户确认取消
3. 删除对应的booking对象

❖ 为了定位要取消的booking对象，必须检查当前显示的每个预约

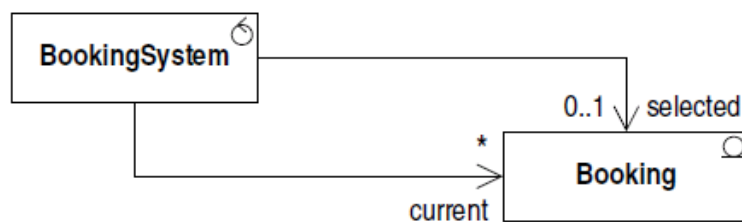
❖ 用角色名区分选中的对象和其他对象

Canceling a Booking



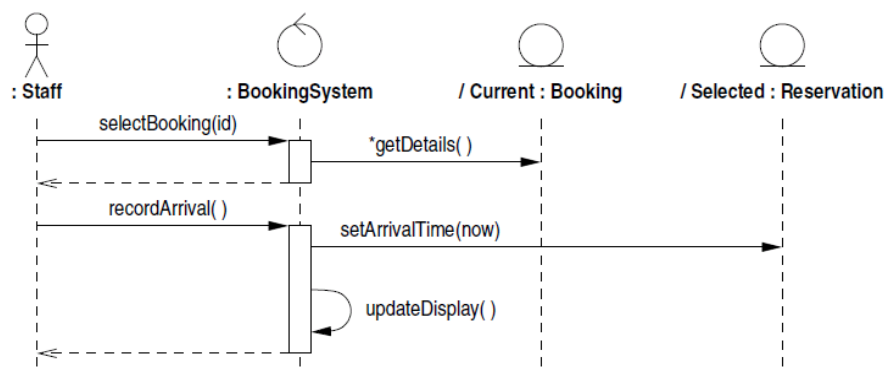
细化领域类图(2)

- ❖ ‘BookingSystem’ 有责任记住选中的预约是哪个
- ❖ 增加一个关联来记录选中的预约



记录到达

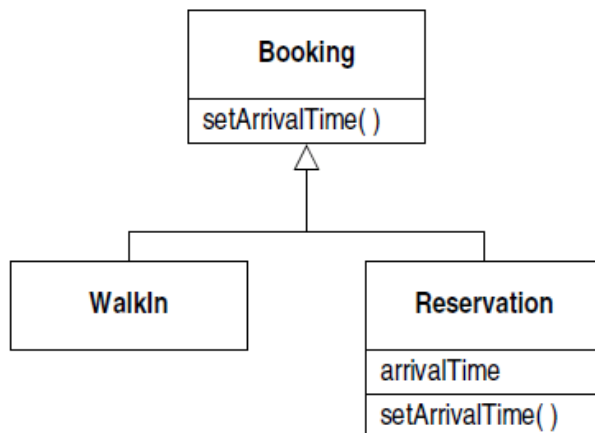
- ❖ 选定的booking必须是reservation



类接口设计

- ❖ 应该将到达时间 ‘arrivalTime’ 存储在哪里?
 - walk-in booking的 ‘arrival time’ 和它的开始时间相同
- ❖ ‘setArrivalTime’ 操作应该在 Booking类中还是 Reservation类 中定义?
 - 这个操作对walk-in不适用
 - 但是我们想尽可能地对所有booking保持一个公共接口
- ❖ 在 ‘Booking’ class中定义操作
 - 默认实现什么都不做
 - 在 ‘Reservation’ class中覆盖操作

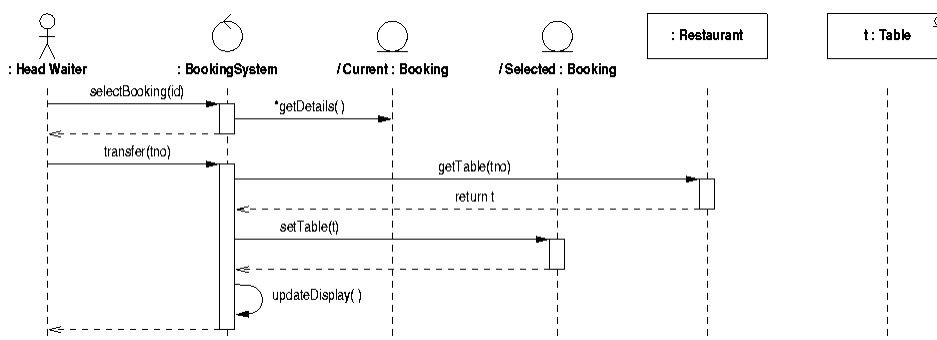
细化类层次



调换餐桌

- ❖ 调换餐桌用例的结构可能看起来更复杂一些
 - 如果我们设想通过拖放booking进行调换，那么用户会收到和鼠标事件对应的很多消息，看起来可能似乎应该在用例实现中反应出来
- ❖ 让应用层的类负责特定UI设备细节并不是一个好想法
- ❖ 在分析层，这个用例最好用单个系统消息建模
 - 为当前选中的预约提供一个新的餐桌标识号与之相关
 - 这种方法保留了用例的基本功能，但是将UI的灵活性留给了后面的设计
- ❖ 这样，Table transfer用例的结构和Record arrival相似

Table Transfer





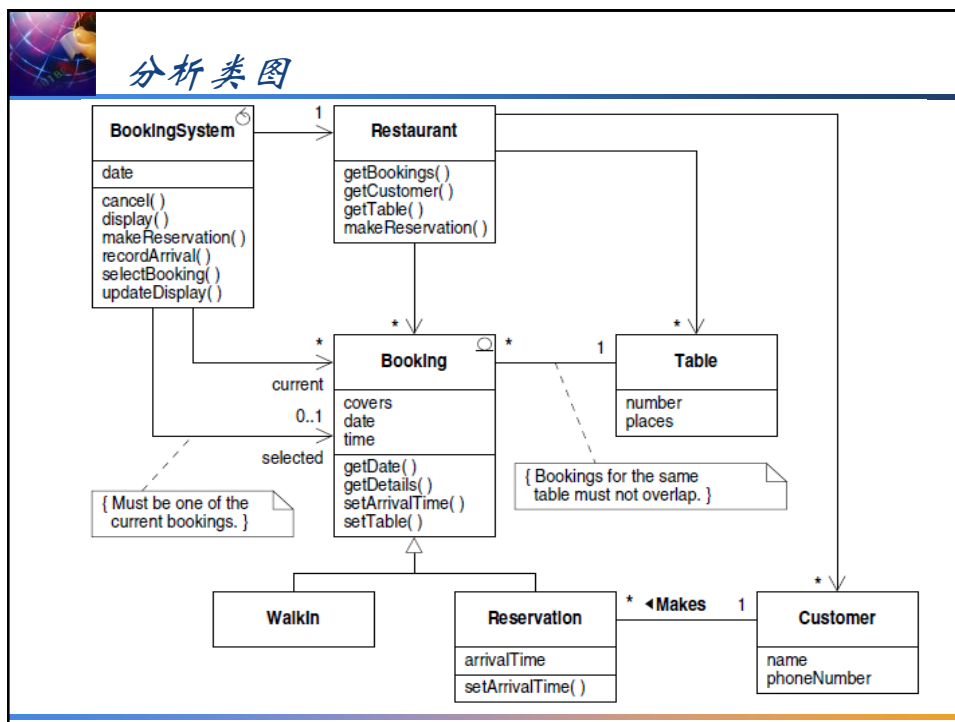
完成分析模型

- ❖ 我们只详细地考虑了每个用例的基本事件路径
- ❖ 在实现可选和异常动作序列时，可以应用同样的原则，这些会在模型中加入一些新东西
- ❖ 一般而言，对每个事件路径用不同的顺序图描述会比较清晰



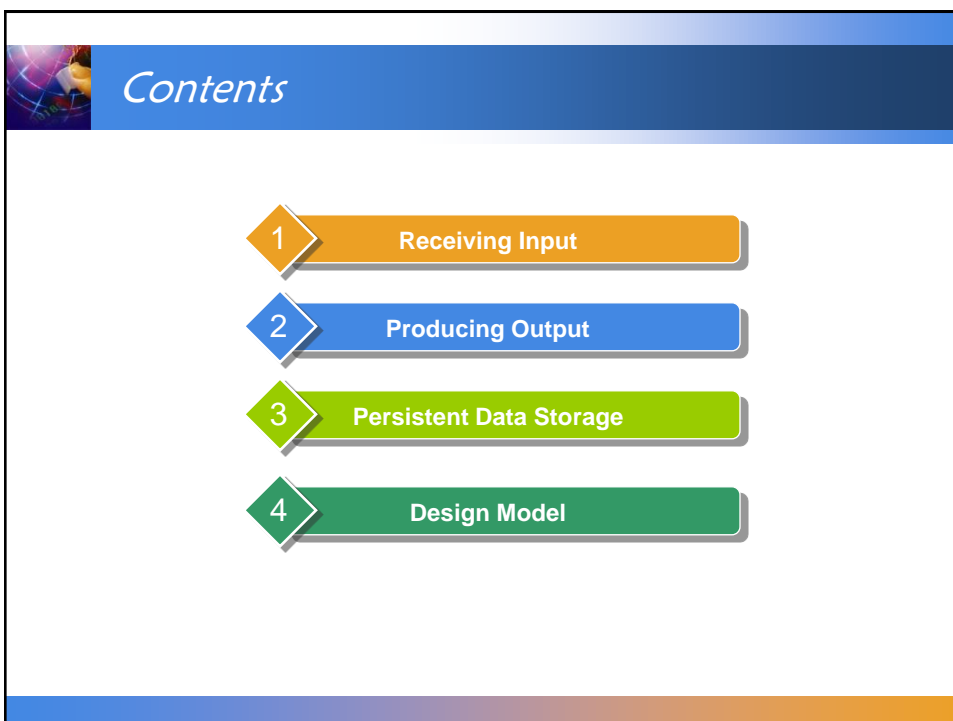
完成分析模型


- ❖ 下面的图显示了系统的类图
 - 包括了在用例实现过程中做出的决策和得出的信息
 - 还包含来自领域模型的信息，例如customer, table 和 booking类以及相关的约束
 - 如果在类的实例之间要传递消息，那么关联在消息发送的方向应该是可导航的



Analysis Summary


- ❖ 分析活动最后得到
 - 一组用例实现
 - 细化的类图
 - 系统架构描述





设计

- ❖ 分析展现了如何在应用层实现业务功能
- ❖ 设计将这个层次的建模层扩展到了其他层
- ❖ 假设预约系统将作为一个桌面应用程序实现
 - 单用户
 - 常规输入和输出设备



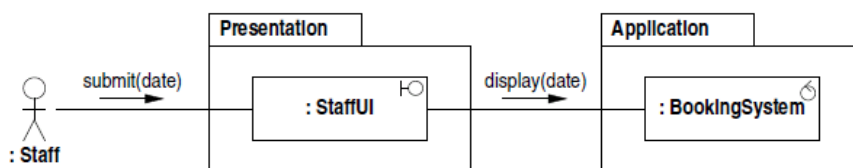
获取用户输入

- ❖ 系统消息作为到应用层中的controller的系统消息描述
- ❖ 事实上这些消息在表示层中已经进行了“预处理”
- ❖ 用表示层中的边界对象对系统的用户界面建模
 - StaffUI
 - 这个类负责和输入设备交互

显示预约

❖ 假设是在对话框中输入日期

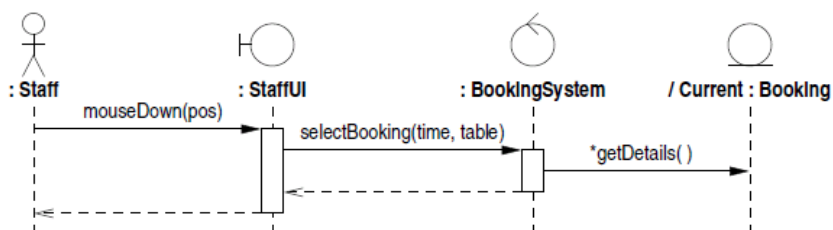
- 不需要对标准对话框的功能细节建模
- ‘submit’ 消息对按下对话框的 ‘OK’ 按钮建模



选择预约

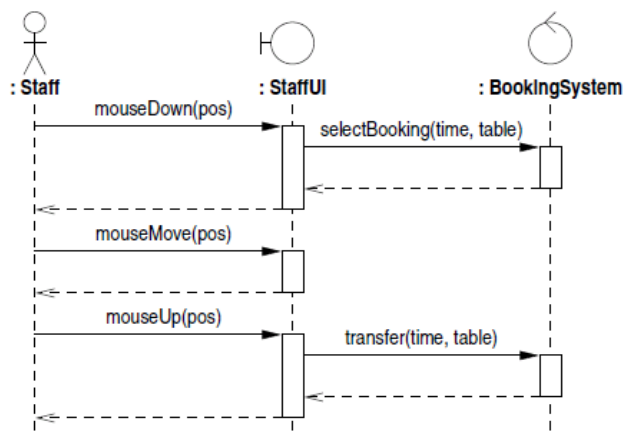
❖ 用相似的方式处理鼠标事件

- 用户界面对象将鼠标事件翻译为系统消息
- ‘time’ 和 ‘table’ 可以从鼠标坐标导出



调换餐桌

- 并不是每个输入事件都需要引发一个系统消息



产生输出

- 输出的责任也被分给用户界面对象
- 只要在应用层发生变化时就应该更新显示
- 问题：如何确保
 1. 在需要时显示总会被更新
 2. 只有在需要时才更新显示



轮询 (Polling)

- ❖ 表示层定义检查应用层的更新
 - 浪费处理时间
 - -区分哪些东西发生了变化的代价比较高
- ❖ 如果应用层的变化能够触发表示层的更新会更好
 - 怎么做到?
 - 在层次结构中, 表示层是不依赖应用层的



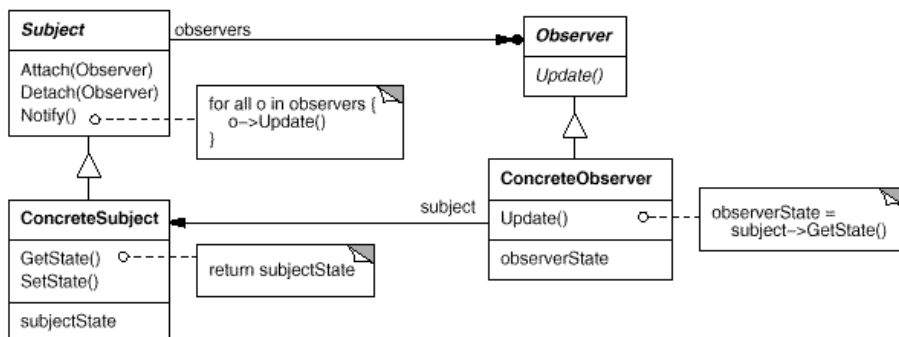
Observer 模式

- ❖ 设计模式是对这种问题的标准解决方案
 - 允许应用层触发表示层中的事件
- ❖ Observer 模式
 - 意图
 - 定义对象之间的one-to-many依赖, 在一个对象的状态改变时, 它的所有依赖者都被通知并自动更新
 - 也叫做Dependents, Publish-Subscribe

Observer 模式——适用性

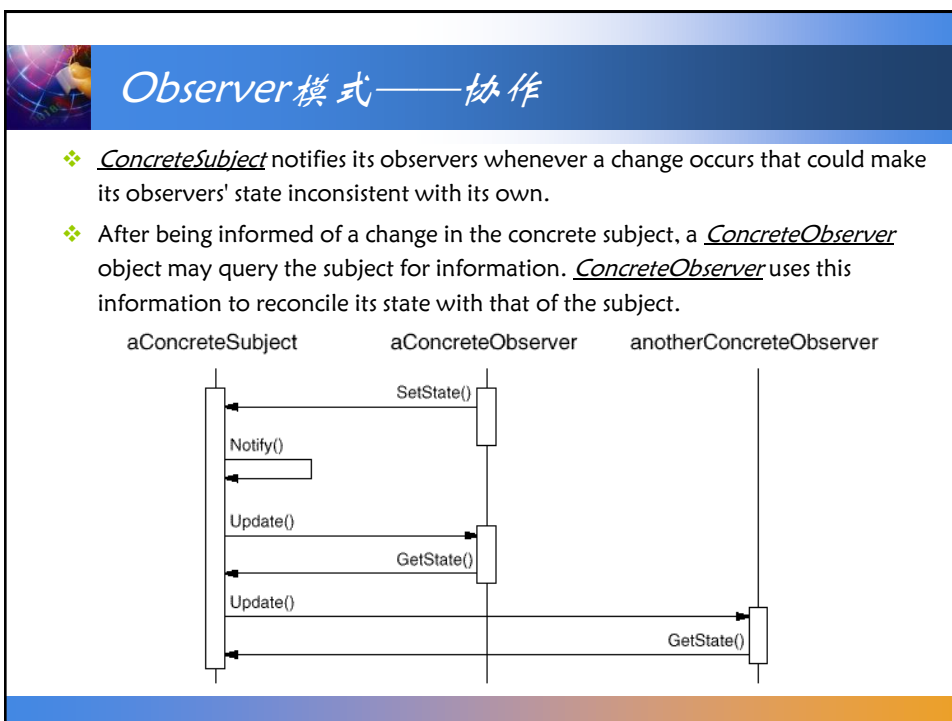
- ❖ 在下面任一种情况下可以使用 Observer 模式
 - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - When a change to one object requires changing others, and you don't know how many objects need to be changed.
 - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer 模式——结构

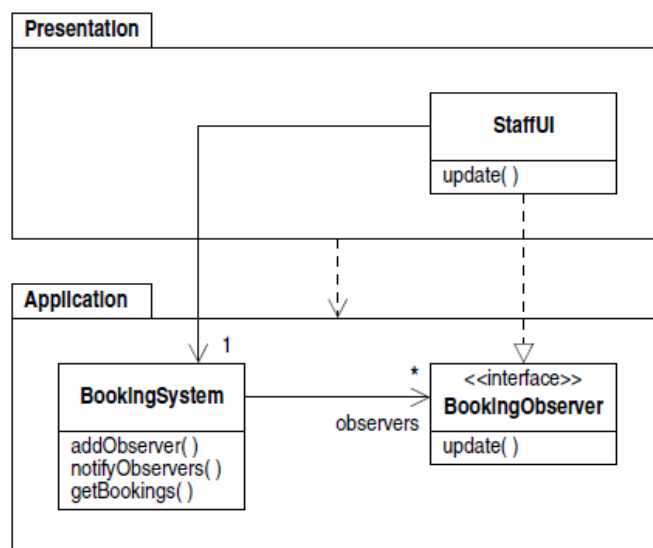


Observer 模式——参与者

- ❖ **Subject**
 - knows its observers. Any number of Observer objects may observe a subject.
 - provides an interface for attaching and detaching Observer objects.
- ❖ **Observer**
 - defines an updating interface for objects that should be notified of changes in a subject.
- ❖ **ConcreteSubject**
 - stores state of interest to ConcreteObserver objects.
 - sends a notification to its observers when its state changes.
- ❖ **ConcreteObserver**
 - maintains a reference to a ConcreteSubject object.
 - stores state that should stay consistent with the subject's.
 - implements the Observer updating interface to keep its state consistent with the subject's.



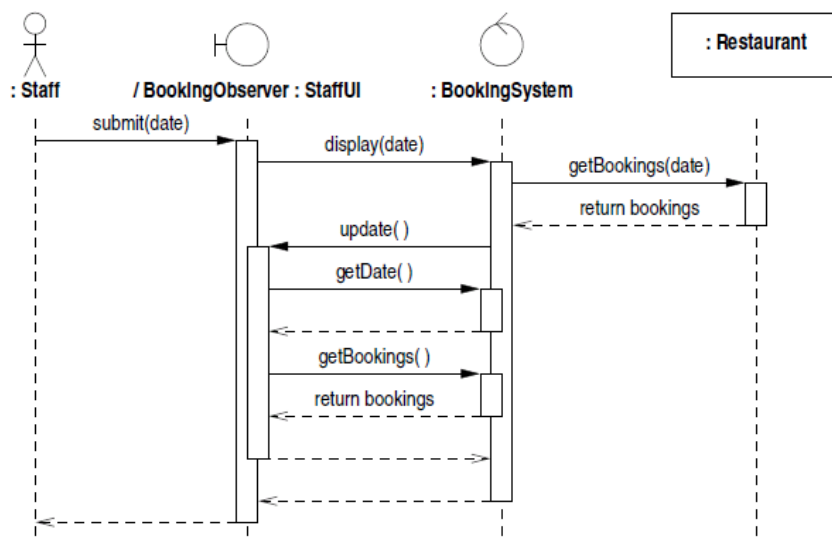
Observer 模式结构



Observer 模式的原理

- ❖ 用户界面类实现booking observer接口
- ❖ BookingSystem维护已注册的observer的列表
 - 不知道它们属于什么类
 - 所以没有向上的依赖
- ❖ 当发生变更时，通知每个observer

显示预约

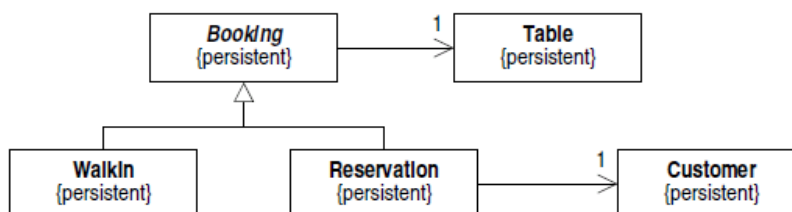


持久数据

- ❖ 大多数系统都需要持久数据
 - 当系统关闭时，持久数据不会丢失
 - 持久数据存储在文件系统或数据库中
- ❖ 面向对象应用程序通常使用关系数据库来提供持久性
- ❖ 设计者需要
 - 识别哪些数据需要持久存储
 - 设计适当的数据库模式 (database schema)

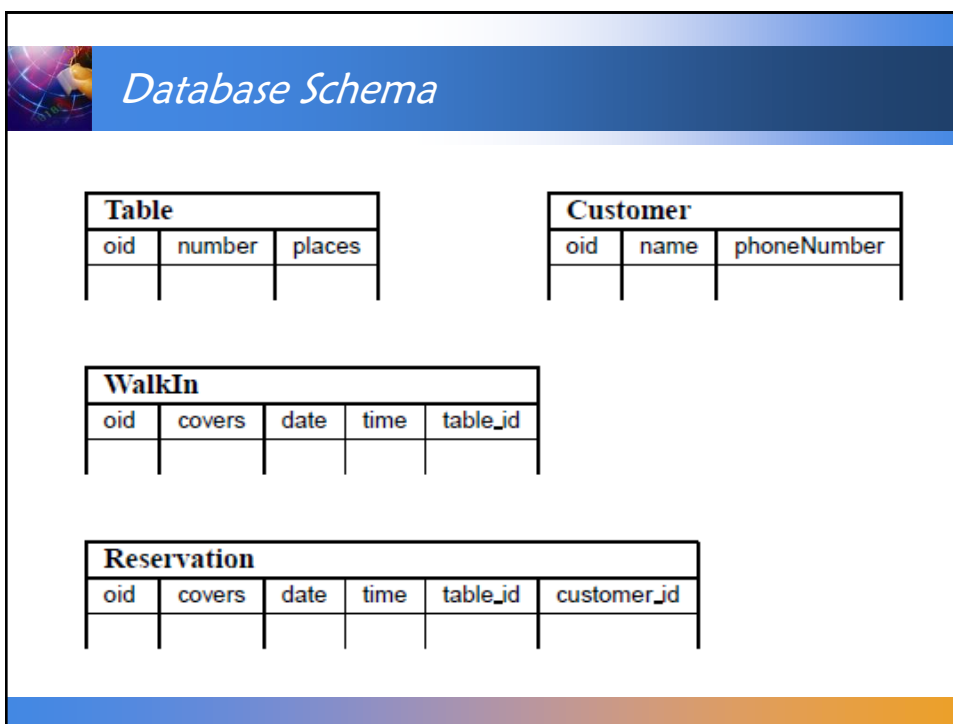
设计持久类

- ❖ 在UML中，类是持久存储的单位
 - 必须保存所有的booking信息
 - 但不用保存当天的预约或当前选中的预约
- ❖ 持久性用标签值（tagged value）表示



创建数据库模式

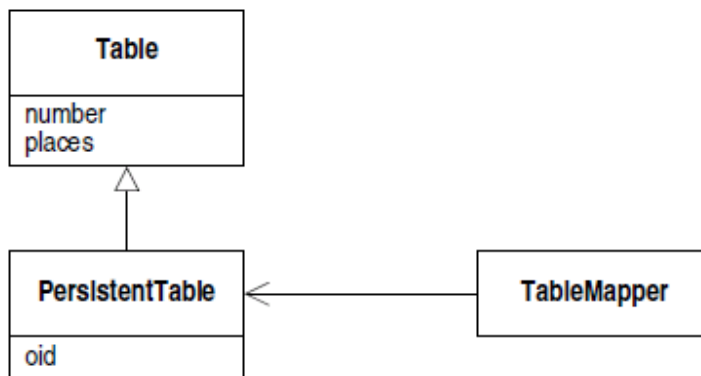
- ❖ 类映射到数据库表
- ❖ 关联是表之间的关系
 - 在每个表中加入显示的对象ID
 - 用对象ID作为外键实现链接
- ❖ 泛化在关系模型中没有等价物
 - 由于‘Booking’是抽象类，只要将具体子类映射为表即可



保存和载入

- ❖ 将数据存入数据库和从数据库中载入数据应该是谁的职责？
 - 使用已有的类可能有低内聚的风险
 - 作为新类的职责
 - 为每个持久类再定义一个映射器 ‘mapper’ 类
- ❖ 在设计模型中加入对象ID
 - 持久性应该排除在领域模型类之外
 - 在子类中加一个 ‘oids’
 - mapper类和持久子类打交道

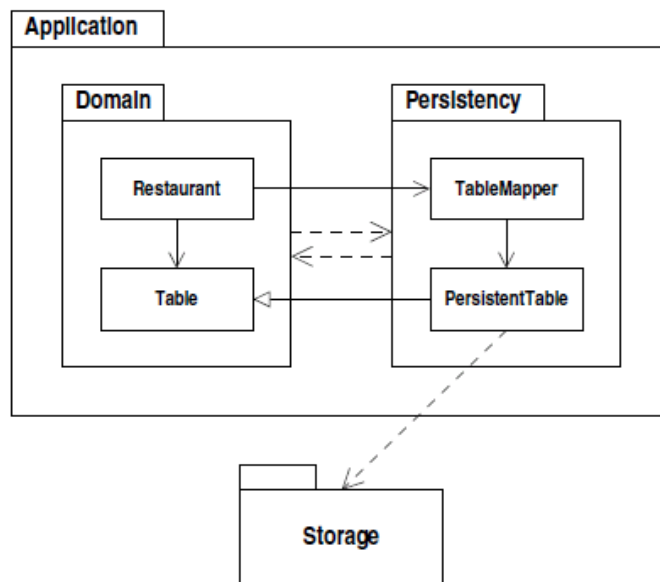
持久性设计



持久性架构

- ❖ 持久子类 and 映射器类依赖它们所支持的类
 - 所以必须放在应用层中
- ❖ ‘Restaurant’ 类依赖映射器类
- ❖ 将应用层分为两个子包
 - 修改 ‘Persistence’ 子包对 ‘Domain’ 子包的影响最小

持久性架构



详细类设计

- ❖ 创建详细的类规格说明
 - 可以用作实现的基础
- ❖ 从细化后的领域模型开始
- ❖ 收集所有实现中的消息，定义类的操作接口
 - 检查冗余和不一致
 - 定义类的操作，说明详细的参数和返回类型

BookingSystem
- date : Date
+ addObserver(o : BookingObserver)
+ cancel()
+ getBookings() : Set(Booking)
+ getDate() : Date
+ makeReservation(d : Date, in : Time, tno : Integer, name : String, phone : String)
+ makeWalkIn(d : Date, in : Time, tno : Integer)
+ notifyObservers()
+ recordArrival()
+ selectBooking(t : Time, tno : Integer)
+ setDate(d : Date)
+ transfer(t : Time, tno : Integer)

Restaurant
getBookings(d : Date) : Set(Booking) getCustomer(name : String, phone : String) : Customer getTable(tno : Integer) : Table makeReservation(d : Date, in : Time, t : Table, c : Customer)

Table
number : Integer places : Integer

Customer
name : String phoneNumber : String

Reservation
arrivalTime : Time setArrivalTime() : Time

WalkIn

Booking
covers : Integer date : Date time : Time getCovers() : Integer getCustomer() : Customer getDate() : Date getTable() : Table getTime() : Time setArrivalTime(t : Time) setTable(t : Table)



对行为建模

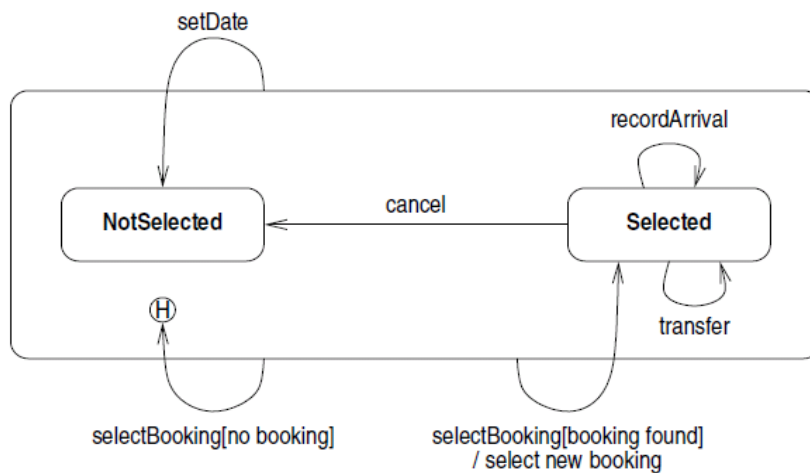
- ❖ 类图显示结构设计
- ❖ 顺序图显示行为
- ❖ 但有些问题没有回答，例如
 - 如果还没有选中booking系统就收到了‘cancel’消息，系统应该做些什么？
 - 如果用户要将一个已取消的预约从一张餐桌移动到另一张，会怎么样？
- ❖ 对这些问题的回答依赖不相关的消息之间的相互作用



状态机 (State Machine)

- ❖ 概括单个类的实例的行为
- ❖ 回答两种问题
 - 对象期望接收到什么消息序列，并做出响应
 - 对象对一个消息的响应依赖对象的历史，即已经接收到的消息
- ❖ 不是每个类都需要状态机图

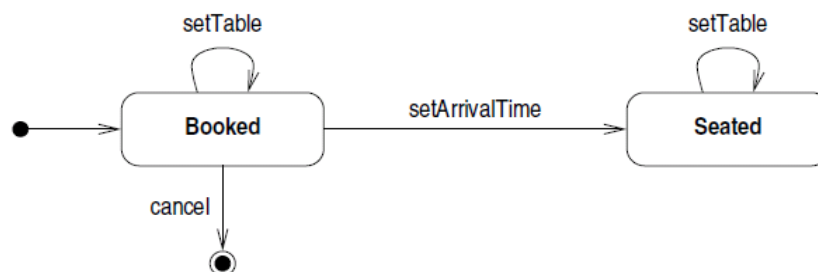
Booking System State Machine Diagram



Reservation State Machine Diagram

❖ Reservations在用餐者到达之后的行为不同

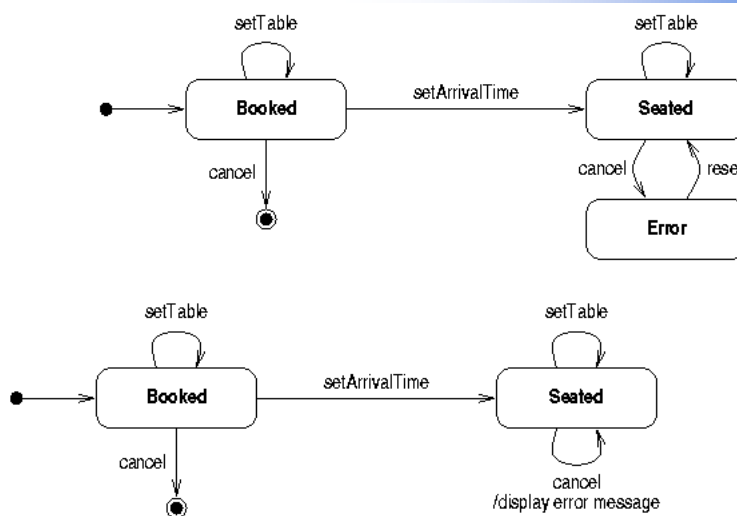
- 不能取消预约




错误处理

- ❖ 对象可能检测到一个事件，但是在当前状态没有匹配的转移
- ❖ UML规定忽略这样的事件
- ❖ 在某些情况下，这可能表示是有错误
 - 为了说明这种情况，可以加一个显式的 ‘Error’ 状态
 - 增加转移，指定系统如何从错误恢复

Error Handling

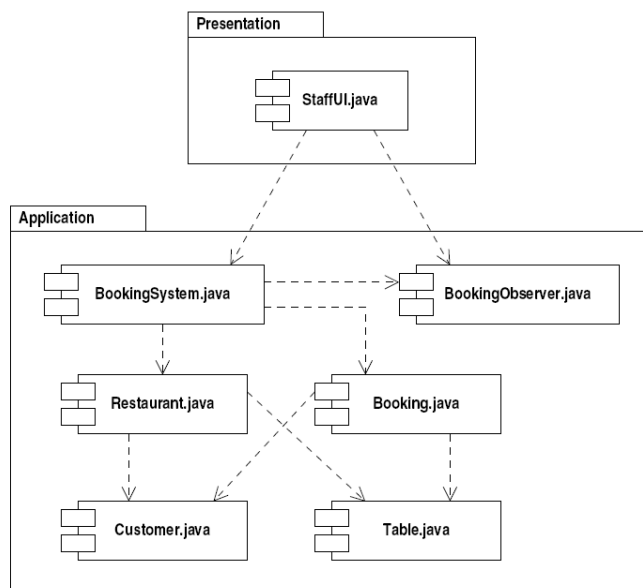




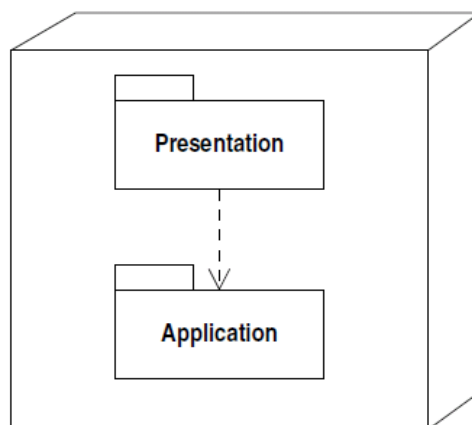
实现

- ❖ 实现是将设计模型转化为可执行代码的过程
 - 很多UML模型元素的实现都有明确的规则
 - 在某种程度上可以自动生成代码
- ❖ UML还有专门说明源代码的物理特性的图
 - 部署图
 - 包图

Booking System的构件图



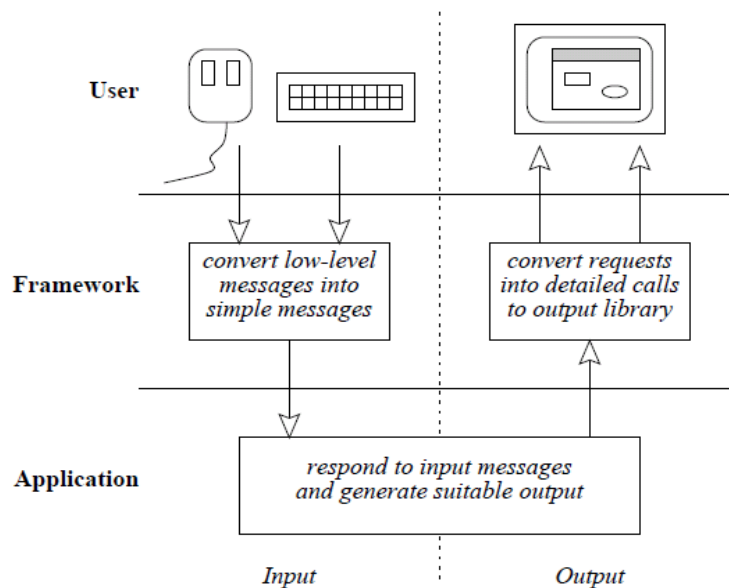
Booking System的部署图



应用框架

- ❖ 面向对象程序通常基于框架 (framework)
- ❖ 框架支持多个应用程序公共的任务
 - 窗口管理
 - 检测输入和输出事件
 - 提供框架应用程序结构
- ❖ 框架提供了相当多的可复用代码

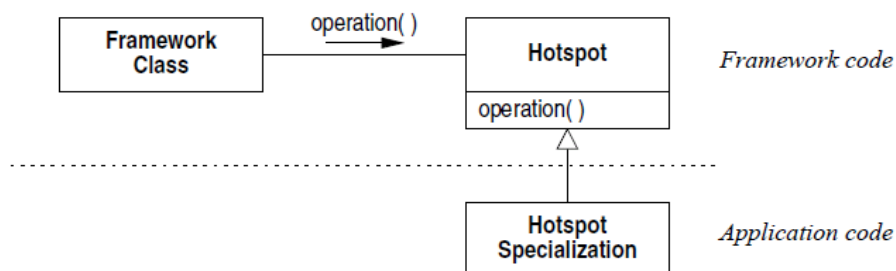
框架的典型使用



代码复用

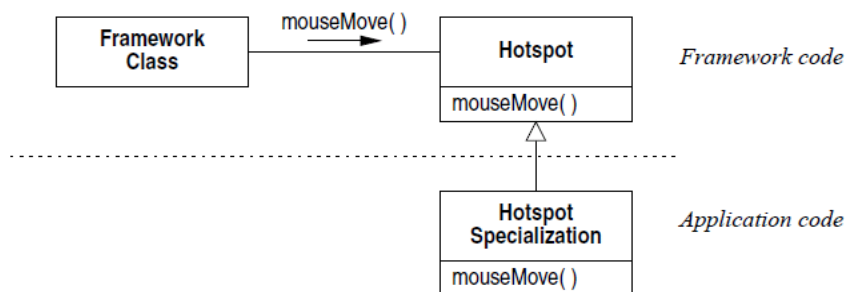
- ❖ 具体应用的代码必须集成到框架中
- ❖ 传统上，代码复用的基础是使用库中定义的构件
- ❖ 面向对象的复用需要应用类特殊化框架中的热点类 (hotspot classes)
- ❖ 应用类
 - 特化框架类
 - 覆盖相关的操作
 - 从框架中接收运行时消息

Hotspots



检测用户输入

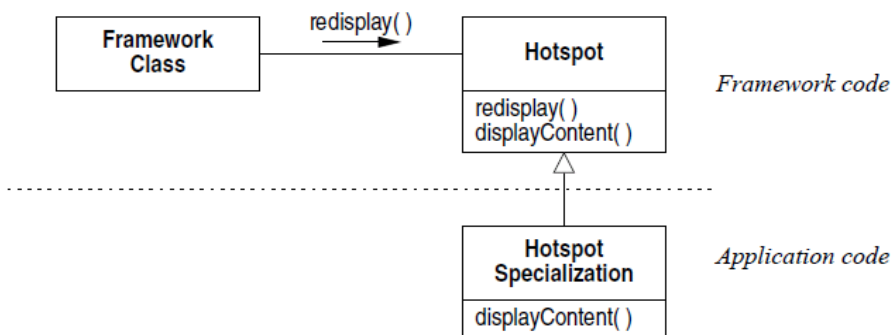
- ❖ 在用户移动鼠标时框架会检测用户输入
- ❖ 应用程序的代码定义对输入的响应



回调 (callback)

- ❖ 有些操作在框架和应用中都要实现
 - 例如，在更新窗口时
 - 框架要绘制边框等窗口元素
 - 应用必须提供窗口的内容
- ❖ 可以提供一个callback或hook函数
 - 例如，框架调用 ‘redisplay’
 - ‘redisplay’ 接着调用 ‘displayContent’
 - 而 ‘displayContent’ 在应用中已覆盖

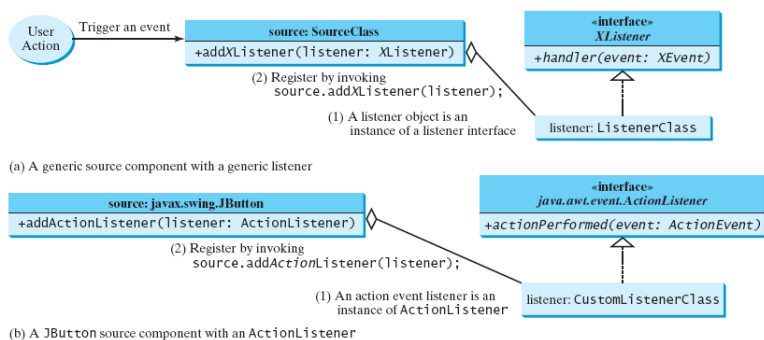
窗口更新



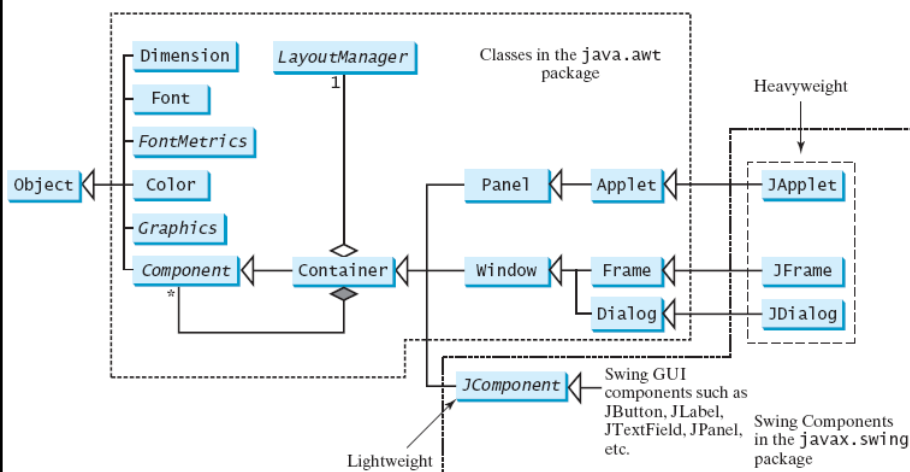
Java GUI 框架

❖ Java GUI API框架支持基于窗口的程序设计

- 用户界面包含各种控件
- listener检测各种感兴趣的事件
- listener可以与响应这些事件的控件关联在一起

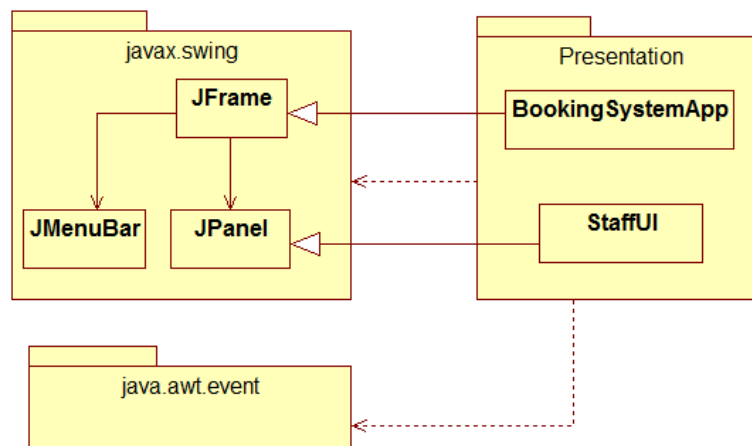



(Some of) Java GUI API in UML



实现Booking System——表示层


❖ Java GUI 框架的使用





代码生成

- ❖ 定义了总体结构之后，就可以实现各个模型元素了
- ❖ 模型元素的实现是基于规则的过程
 - 可以用代码生成工具支持
- ❖ 实现时也有很多可选的策略



实现类

- ❖ 基本映射
 - UML类用语言的class定义
 - 属性用数据成员实现
 - 操作用方法实现
- ❖ 具体语言相关的问题
 - 抽象类的实现
 - 接口的实现
 - 模板的实现

The Booking Class

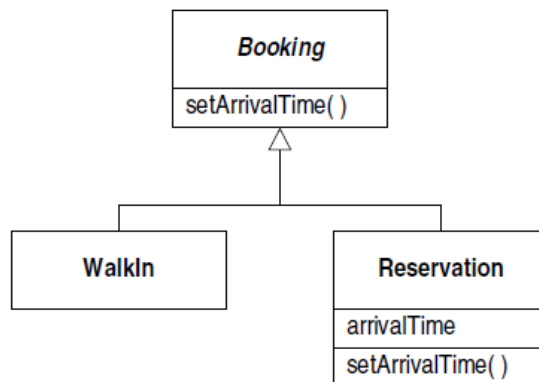
```
public abstract class Booking {
    protected int covers ;
    protected Date date ;
    protected Booking(int c, Date d) {
        covers = c ; Date = d ;
    }
    public Date getDate() { return date; }
    public void setArrivalTime(Time t) { }
    public void setCovers(int c) { covers = c; }
}
```

<i>Booking</i>
covers : Integer date : Date
getCovers(c : Integer) getDate() : Date setArrivalTime(t : Time)

泛化

❖ 与继承相似

- 子类到超类的类型转换
- 属性和操作的继承





泛化的实现

```
public class WalkIn extends Booking
{
    public WalkIn( ... ) { ... }
}

public class Reservation extends Booking
{
    Time arrivalTime ;
    public Reservation( ... ) { ... }
    public void setArrivalTime( Time t ) {
        arrivalTime = t ;
    }
}
```



类的重数

- ❖ 类的实例个数在默认情况下是任意的
- ❖ 类重数限制
 - 用Singleton模式确保 ‘BookingSystem’ 类只有一个实例

BookingSystem¹



Singleton Pattern

```
public class BookingSystem
{
    // Single instance stored and accessed
    // via 'getInstance' method
    private static BookingSystem uniqueInstance
        ;
    public static BookingSystem getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new BookingSystem() ;
        }
        return uniqueInstance ;
    }
    // Private constructor to prevent creation
    // of other instances
    private BookingSystem( ) { ... }
}
```

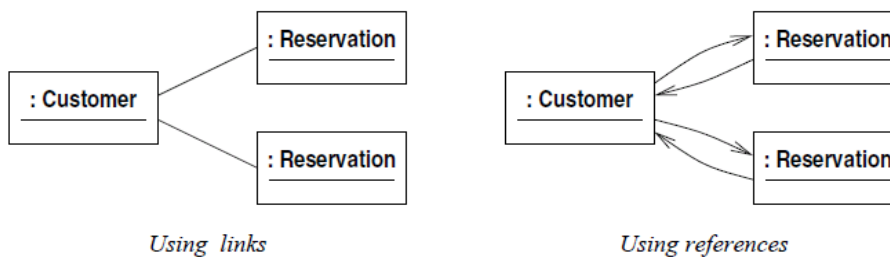


实现关联

- ❖ 链接用引用（指针）实现
 - 记录链接的对象的位置和标识（地址）
 - 允许消息传递
- ❖ 关联可以作为定义适当引用（指针）的数据成员实现

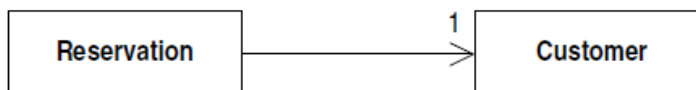
双向性

- ❖ 两个引用可以实现一个链接
 - 消息可以在两个方向传递



单向实现

- ❖ 通常只需要实现关联在一个方向上的导航性



```
public class Reservation {
    private Customer customer ;
    public Reservation( Customer c ) {
        customer = c ;
    }
}
```

重数

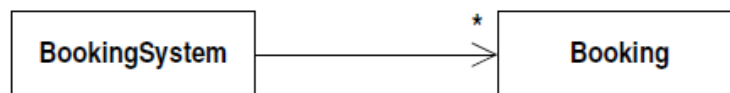
❖ 关联的重数要在运行时检查

- 每个reservation应该链接到恰好一个customer
- 但reservation可能保存一个null引用

```
public Reservation( Customer c ) {
    if ( c == null ) { /* throw
        NullCustomerException */ }
    customer = c ;
}
```

实现 ‘many’ 重数

❖ 大于1的重数要用数据结构保存引用



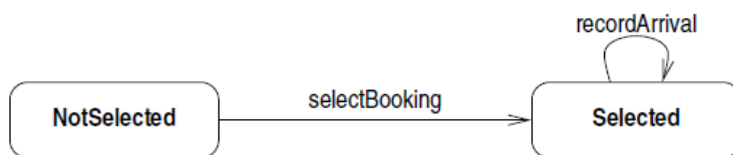
```
public class BookingSystem {
    private Vector current = new Vector( ) ;
    public void addBooking( Booking b ) {
        current.addElement(b) ;
    }
}
```

实现操作

- ❖ 操作用方法实现
- ❖ 利用含有相应消息的顺序图得出操作的实现
- ❖ 如果有状态机，也可以用来组织类的操作实现

实现状态机

- ❖ 状态机描述对象的动态行为
 - 可以反映在类的操作中
 - 例如BookingSystem 的状态机





实现状态

❖ 将状态定义为常量:


```
public class BookingSystem
{
    final static int NotSelected = 0 ;
    final static int Selected = 1 ;
    int state ;
    BookingSystem( ) {
        state = NotSelected ;
    }
}
```



依赖状态的行为

❖ 在每个操作中检查当前状态

```
public void operation( ) {
    switch (state) {
        case NotSelected:
            // Specific actions for 'Not Selected'
            state
            break ;
        case Selected:
            // Specific actions for 'Selected' state
            break;
    }
}
```



课程实践作业

❖ 建立下面系统的用例模型、领域模型、分析模型和设计模型

- ATM系统
- 校园一卡通管理系统
- 网络书店管理系统
- 图书馆管理系统
- 某宾馆预约住宿管理系统
- 某商场会员卡管理系统
- 某饮用水公司的电话预约送水管理系统



The End !