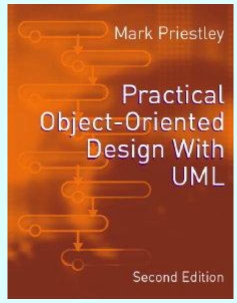
# PRACTICAL OBJECT-ORIENTED DESIGN WITH UML 2e




Chapter 6:
**Restaurant System: Design**







## Design

- Analysis shows how business functions can be implemented in the application layer
- Design extends this level of modelling to the other layers
- We assume the booking system will be implemented as a desktop application, ie:
  - single user
  - normal input and output devices

# Getting Input From The User

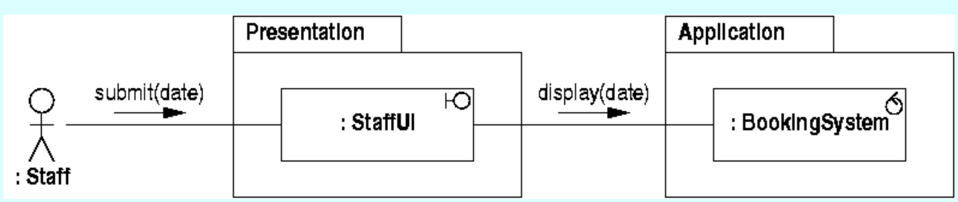
- System messages have been shown arriving at the controller in the application layer
- In fact these will have been ‘pre-processed’ in presentation layer
- A *boundary* object in the presentation layer models the system's user interface
- This has the responsibility for interacting with input devices





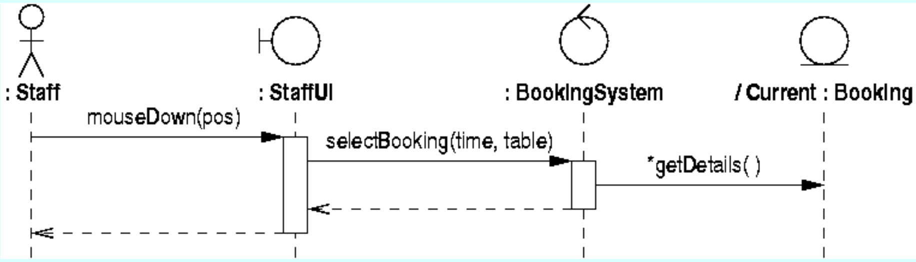
# Display Bookings

- Assume date is entered in a dialogue box
  - don't need to model standard dialogue box functionality in detail
  - ‘submit’ message models pressing ‘OK’ button



# Selecting a Booking

- Mouse events are handled similarly
  - user interface object translates mouse events into system messages
  - ‘time’ and ‘table’ derived from mouse coordinates



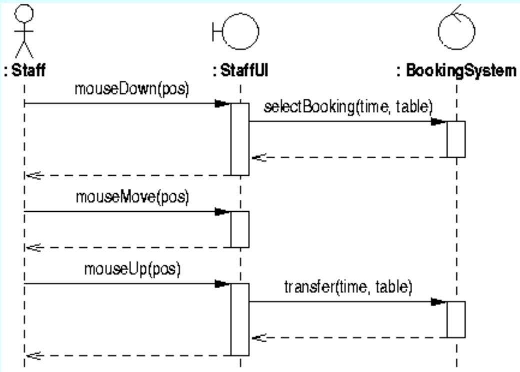






# Table Transfer

- Not every input event need give rise to a system message



# Producing Output

- Output also assigned as a responsibility of the user interface object
- The display should be updated whenever the application layer changes
- Problem: how to ensure that the display is:
  - *always* updated when it needs to be
  - *only* updated when it needs to be






# Polling

- The presentation layer would periodically check the application layer for updates
  - wasteful of processing time
  - expensive to tell if something has changed
- Better if the application layer triggered updates in the presentation layer
  - but how? In the layered architecture, the presentation layer is supposed to be independent of the application layer

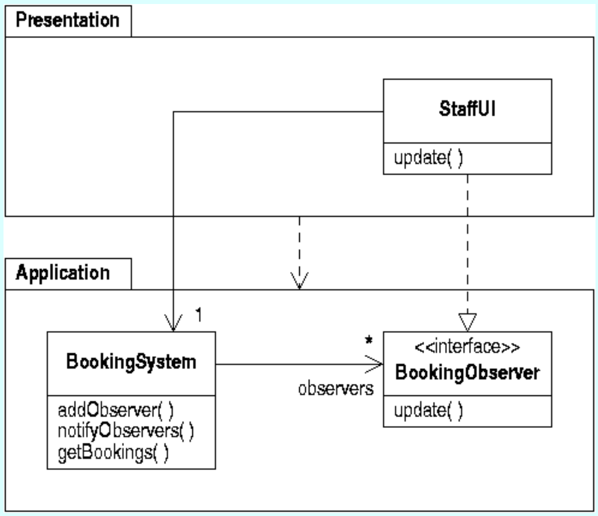
# Observer Pattern

- *Design patterns* are standard solutions to problems like this
- In particular, the *Observer* pattern:
  - allows changes to one object (eg application) to be communicated to others (eg presentation)
  - without assuming what the other objects are
- This will allow the application layer to trigger events in the presentation layer





# Observer Pattern Structure



# Interfaces in UML

- Interfaces are represented as stereotyped classes
  - they have operations but no attributes
- Classes can *realize* interfaces
  - shown as a dashed arrow with an open arrowhead from the class to the interface
  - this means the class must implement all the operations defined in the interface

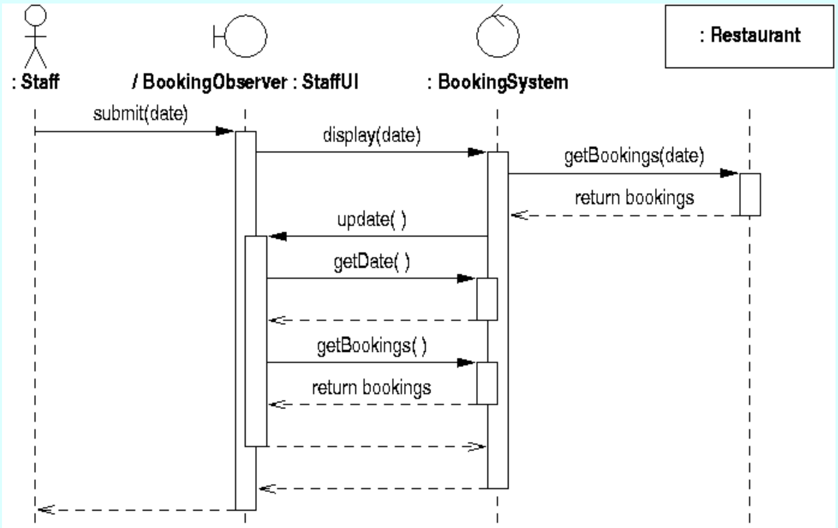


Practical Object-Oriented Design with UML 2e

# Observer Pattern Rationale

- The user interface class implements the booking observer interface
- The booking system maintains a list of registered *observers*
  - but it doesn't know what class they belong to
  - so there is no 'upwards' dependency
- Each observer is notified whenever a change takes place

# Displaying bookings










# Explanation

- The interface implemented by the 'StaffUI' object is shown as a *role*
- When something changes, the 'update' messages is sent to the user interface
- It then gets updated information from the booking system
  - the simplest option is to redisplay everything
  - this can be optimized later if necessary

# Persistent Data

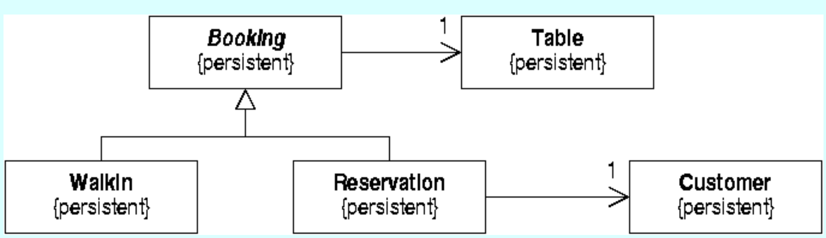
- Most systems need *persistent* data which is
  - not lost when the system closes down
  - stored in file system or database
- Object-oriented applications often use relational databases to provide persistence
- Designer needs to:
  - identify what data needs to be persistent
  - design a suitable database schema






# Designating Persistent Classes

- In UML classes are the unit of persistence
  - we must save all booking information
  - but not eg current date, or selected bookings
- Persistence is shown using a *tagged value*



# Creating a Database Schema

- Classes map to tables
- Associations are relationships between tables, so:
  - add explicit object IDs to each table
  - use these as foreign keys to implement links
- Generalization has no relational equivalent
  - as 'Booking' is an abstract class, simply map concrete subclasses as tables







# Database Schema

- From this we can derive a simple schema

**Table**

| oid | number | places |
|---|---|---|
| | | |

**Customer**

| oid | name | phoneNumber |
|---|---|---|
| | | |

**WalkIn**

| oid | covers | date | time | table_id |
|---|---|---|---|---|
| | | | | |

**Reservation**

| oid | covers | date | time | table_id | customer_id |
|---|---|---|---|---|---|
| | | | | | |

# Saving and Loading

- Whose responsibility is it to save objects and load them from the database?
  - using existing classes risks low cohesion
  - make this the responsiblity of a new class
  - define a ‘mapper’ class for each persistent class
- Include object IDs in design model
  - keep persistency out of domain model classes
  - add ‘oids’ in a subclass

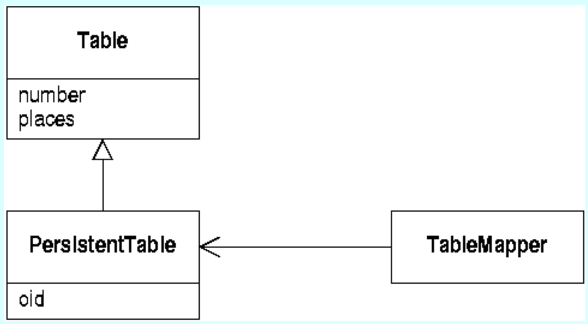




Practical Object-Oriented Design with UML 2e

# Design for Persistency

- The mapper classes deal with the persistent subclasses



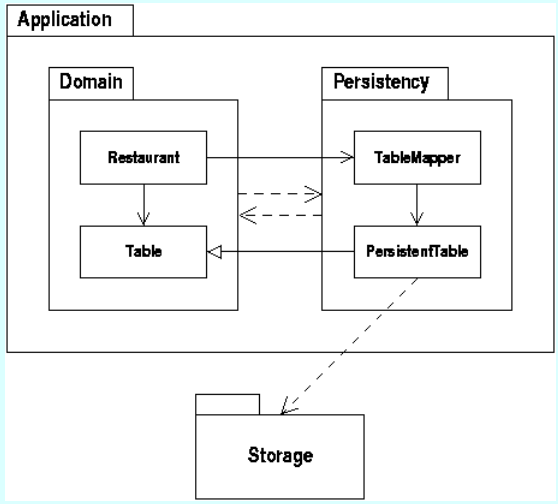
# Persistency Architecture

- Persistent subclasses and mapper classes depend on the class they are supporting
  – so they must be in the application layer
- But the 'Restaurant' class is dependent on mapper classes
- Split application layer into two subpackages
  – change to 'Persistency' subpackage has minimal effect on 'Domain' subpackage






# Persistency Architecture



# Detailed Class Design

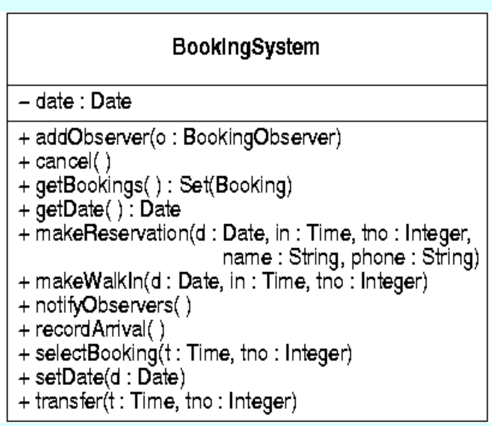
- Create a detailed class specification that could be used as a basis for implementation
- Start with refined domain model
- Collect messages from all realizations
  - check redundancy, inconsistency etc
  - this defines the operation interface of class
  - specify detailed parameters and return types







# The Booking System Class



# Modelling Behaviour

- Class diagrams show structural design
- Sequence diagrams show behaviour
- But some questions are not answered, eg:
  – what should the system do if a ‘cancel’ message is received before a booking is selected?
  – what happens if the user tries to move a cancelled booking from one table to another?
- These depend on the interaction between separate messages






# Statecharts

- Summarize the behaviour of instances of a single class
- They answer two types of question:
  – what *sequences* of messages the objects are expected to receive and respond to
  – how an object's response to a message depends on its *history*, ie the messages that have already been received
- Not every class will require a statechart

# Booking System Statechart

- The behaviour of the booking system is different if a booking is selected
- This suggests that it has (at least) two states



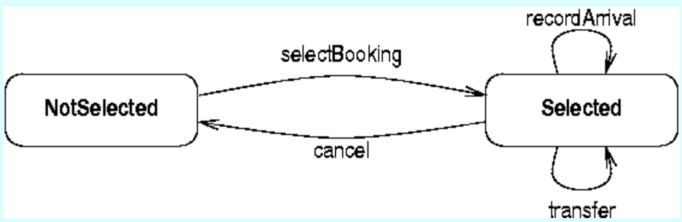

Practical Object-Oriented Design with UML 2e

# Basic Statechart Properties

- At any given time, an object is in exactly one of a number of *states*
- When a message is received, a *transition* will *fire* if:
  - there is a transition leaving the current state
  - labelled with an *event* corresponding to the message
- The object may then end up in a different state

# Operations on Bookings

- Further transitions can be added for all the events an object can detect



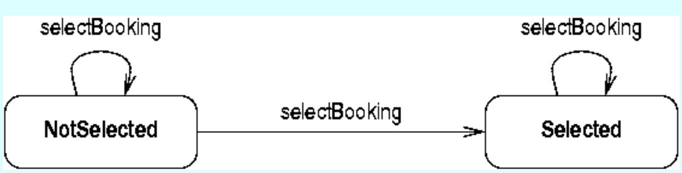





Practical Object-Oriented Design with UML 2e

# Nondeterminism

- Sometimes an event can have more than one transition
- For example, the user may try to select a booking with the mouse over empty space
  - nothing will be selected and the state will not change



# Guard Conditions

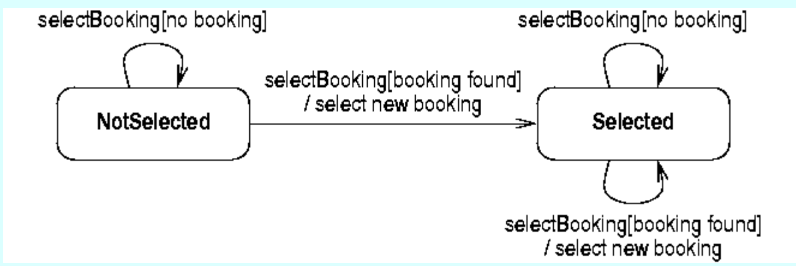
- Nondeterminism can be removed using *guard conditions*
  - these are Boolean expressions stating when each transition will fire




Practical Object-Oriented Design with UML 2e

# Actions

- Statecharts can show what an object does in response to a particular event
- These are shown as *actions* attached to the relevant transition



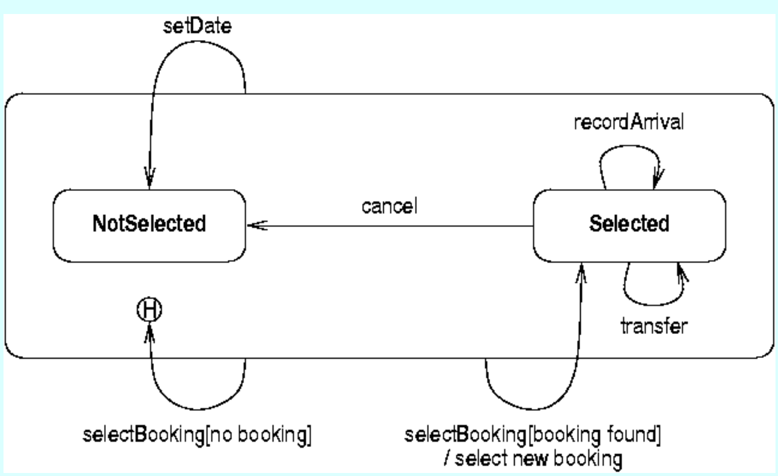
# Composite States

- Composite states can simplify diagrams
  - they define properties shared by all the 'nested' states
- Transitions can freely cross the boundary
- Transitions *from* a composite state apply to all the nested states
- History states 'remember' the most recent nested state
  - a transition to a history state goes to that state



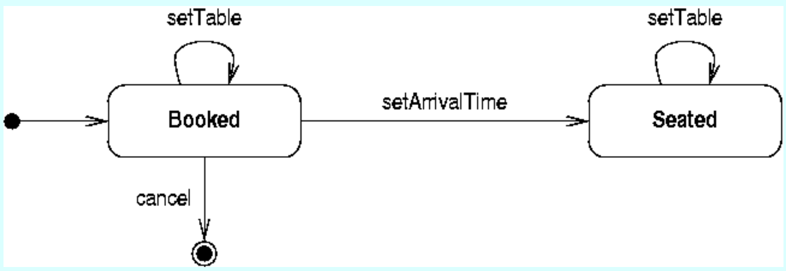
Practical Object-Oriented Design with UML 2e

# Booking System Statechart



# Reservation Statechart

- Reservations have different behaviour after the diners have arrived
  – can't then cancel the reservation








# Initial and Final States

- Initial states model object creation
  – a transition from an initial state corresponds to a constructor
  – but: initial states inside composites indicate what state a transition ending at the composite will end up at
- Final states model object destruction
  – once an object reaches a final state it cannot detect events

# Error Handling

- An object may detect an event when there is no matching transition from its current state
- UML specifies that event is simply ignored
- In some cases, this indicates an error
  - to show this, add an explicit ‘Error’ state
  - add transitions to specify how the system recovers from the error