Component interaction

- The component identification gives us an initial set of interfaces and components with which to work.
- Now we will decide how the components will work together to deliver the required functionality.

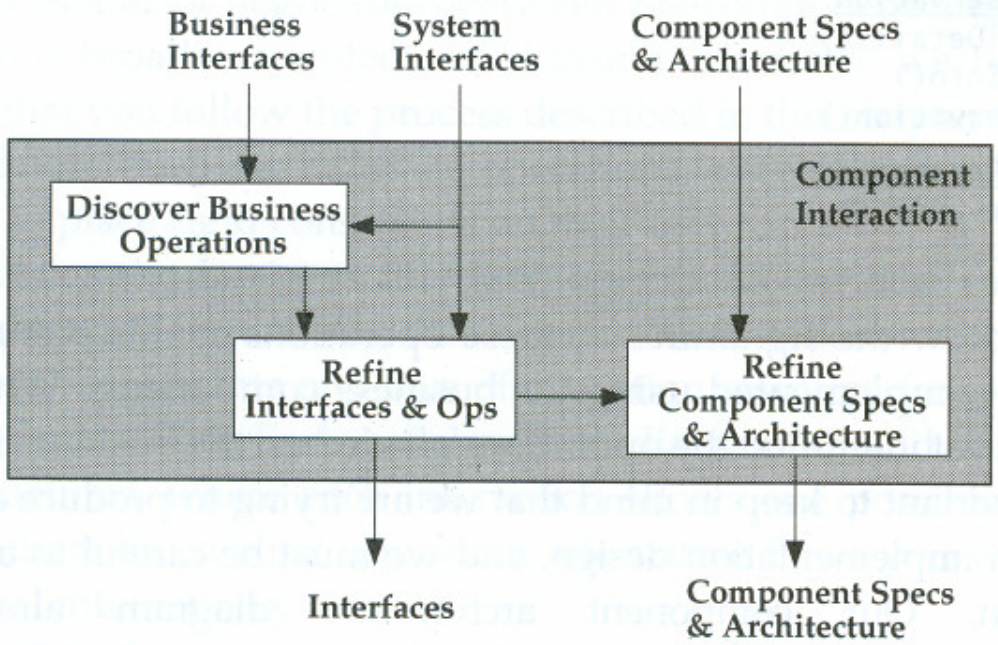


Figure 6.1 The component interaction stage of the specification workflow

Discovering business interfaces I

- We have identified the operations of the system interfaces.
- Example: Interface IMakeReservation has the operations getHotelDetails(), getRoomInfo(), and makeReservation().
- We do not know the signatures of these operations at this point, nor how they will be implemented using business components.
- We haven't even identified the operations needed on the business interfaces.
- Our component architecture diagram tells implementers of ReservationSystem that they must use the ICustomerMgt and IHotelMgt interfaces.

Discovering business interfaces II

Procedure for discovering business operations:

- Take each system interface operation and draw one or more collaboration diagrams that trace any constraints on flows of execution resulting from an invocation of that operation.
- Each collaboration diagram should show one or more interactions, where each interaction shows one possible execution flow.
- So if there are several important flows, one will need to draw several interactions.

getHotelDetails() I

- Input: string to be used as a partial match against the hotel names.
- Output: collection of hotel details

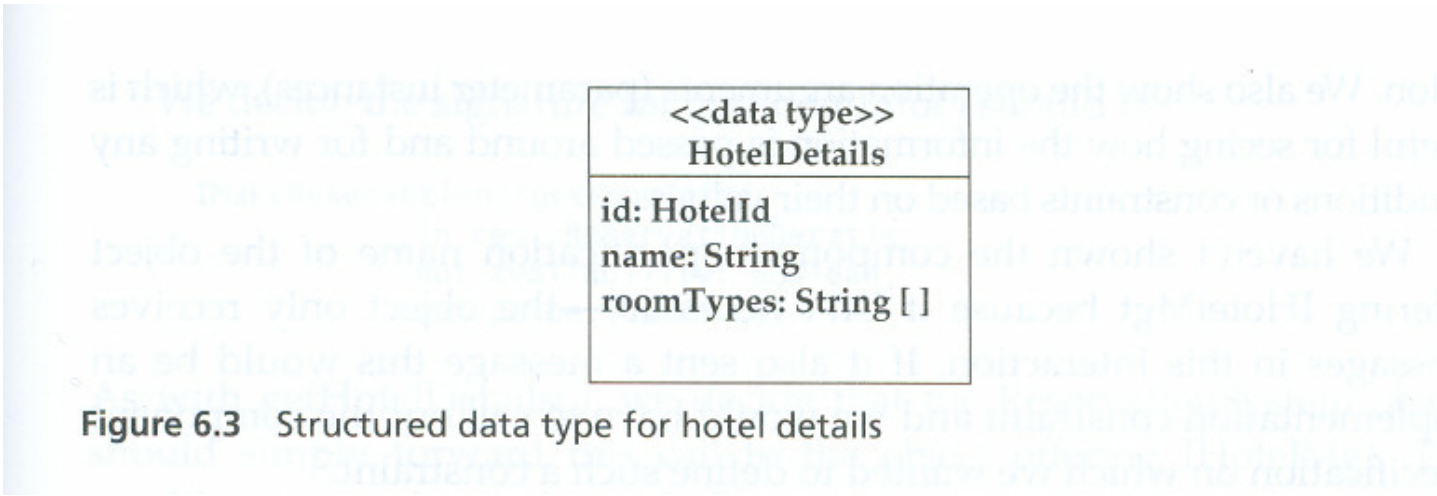


Figure 6.3 Structured data type for hotel details

```
IMakeReservation::getHotelDetails(
    in match:  String):  HotelDetails []
```

getHotelDetails() II

Collaboration diagram:

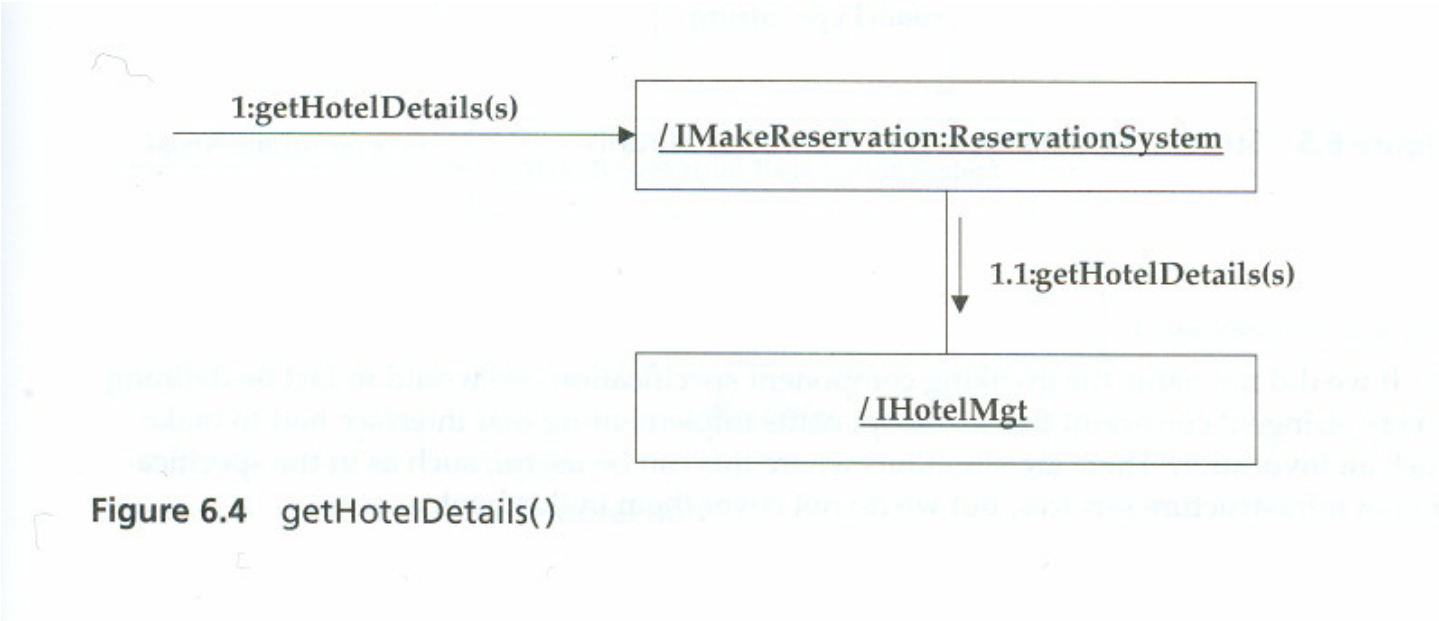


Figure 6.4 getHotelDetails()

(Notation: objectname/rolename:classifiername)

getRoomInfo()

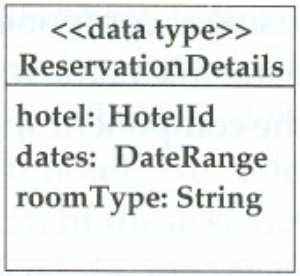


Figure 6.5 Structured data type for reservation details

```
IMakeReservation::getRoomInfo(
in res:  ReservationDetails,
out availability:  Boolean, out price:  Currency)
```

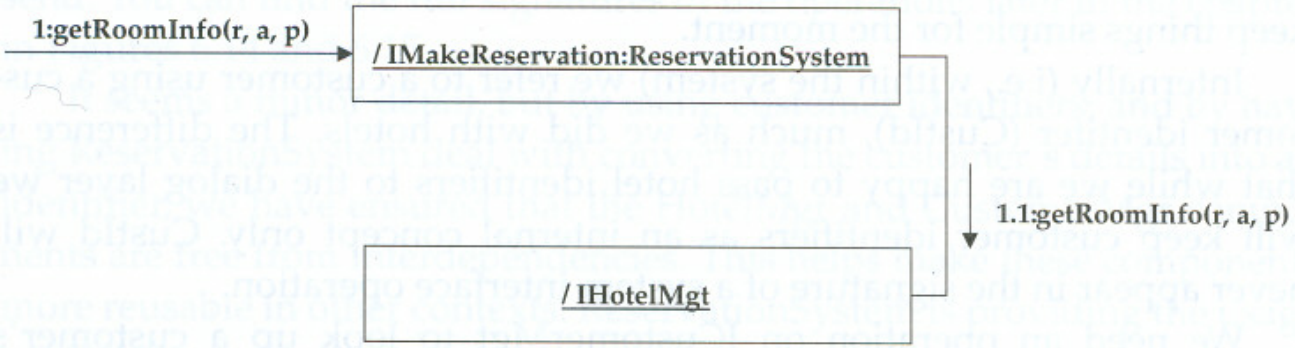


Figure 6.6 getRoomInfo() interaction

makeReservation(): breaking dependencies I

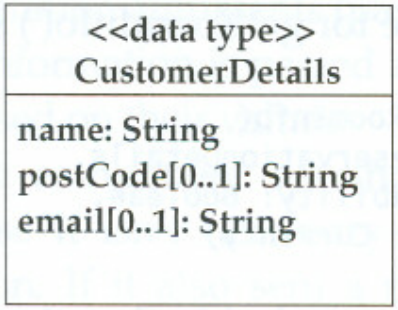


Figure 6.7 Structured data type for customer details

```
IMakeReservation::makeReservation(
   in res:  ReservationDetails,
   in cus:  CustomerDetails, out resRef:  String):
                                          Integer
```

where the return value indicates the outcome of the operation

- 0: Success.
- 1: Customer does not exist, no new record could be created, because post code and/or e-mail address were not provided.

makeReservation(): breaking dependencies II

- 2: No post code was provided, and the name matches more than one customer.

We need an operation on ICustomerMgt to look up a customer's details and return his or her CustId, so we invent one:

```
ICustomerMgt::getCustomerMatching(
   in cusD: CustomerDetails,
   out cusID: CustId):  Integer
```

where 0: success; 1: customer does not exist; 2: as above.

makeReservation(): breaking dependencies III

Which of our components is going to call that operation?

- The HotelMgr component is responsible for storing the association between reservations and customers.
- The HotelMgr and CustomerMgr components are independent of each other!
- Therefore, we cannot let the ReservationSystem component forward the makeReservation() call to the HotelMgr and let it get on with it, because then HotelMgr would have to use CustomerMgr.
- Instead, the ReservationSystem is going to have to do this.

makeReservation(): breaking dependencies IV

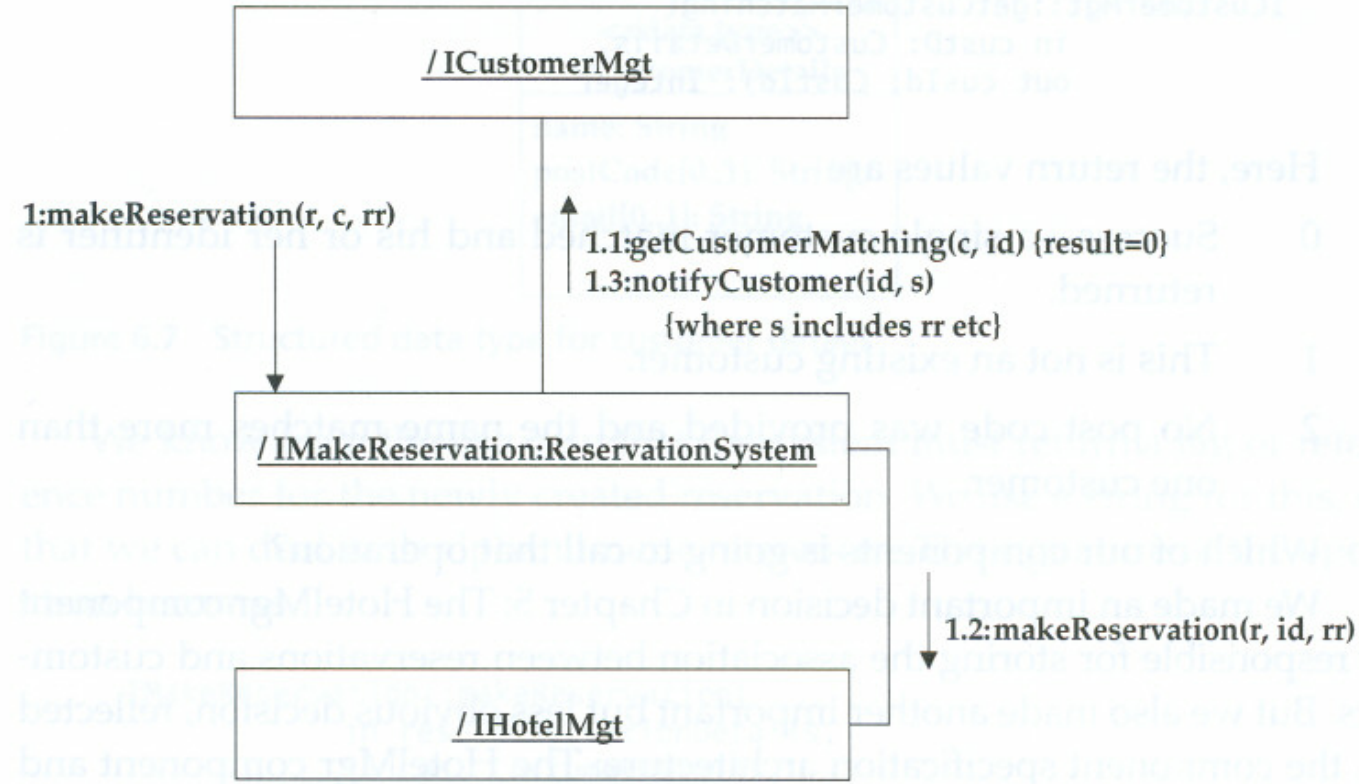


Figure 6.8 makeReservation() interaction (existing customer)

Maintaining referential integrity

- We haven't said how many component objects there will be at runtime.
- Example: ReservationSystem will always use the same business component objects.
- Expressed using a component specification diagram.

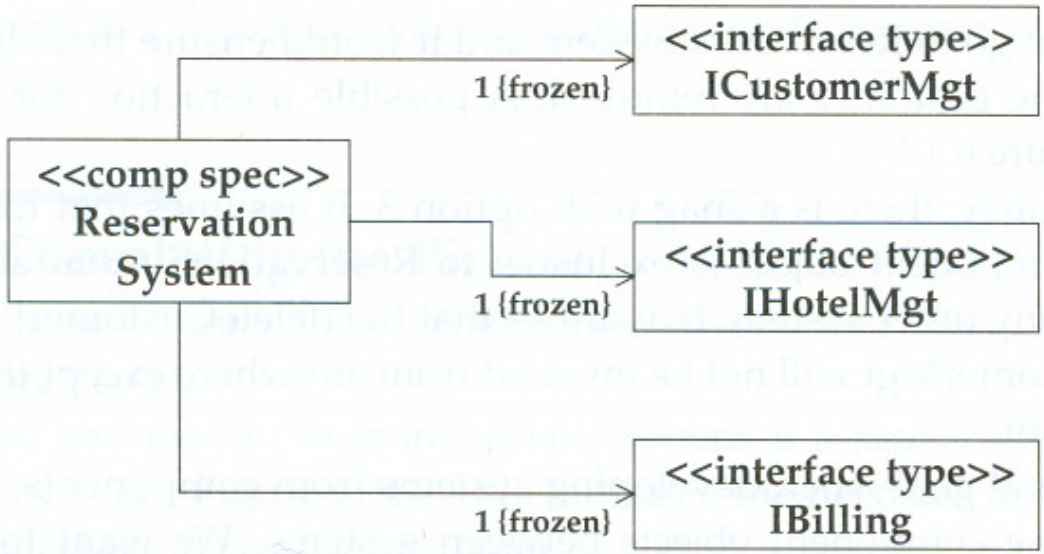


Figure 6.9 Constraints on the component object architecture

Controlling intercomponent references I

Options for allocating responsibility that intercomponent references are valid (example: deletion of a customer):

1. Allocate responsibility to the component object storing the reference.
   Example: make sure that all requests to delete customers are sent to the HotelMgr component.
2. Allocate responsibility to the component object that owns the target of the reference.
   Example: this would be CustomerMgr.
3. Allocate responsibility to a third object, usually higher up in the call chain.
   Example: ReservationSystem.
4. Permit, and tolerate, references to become invalid.
5. Disallow the deletion of information.

Controlling intercomponent references II

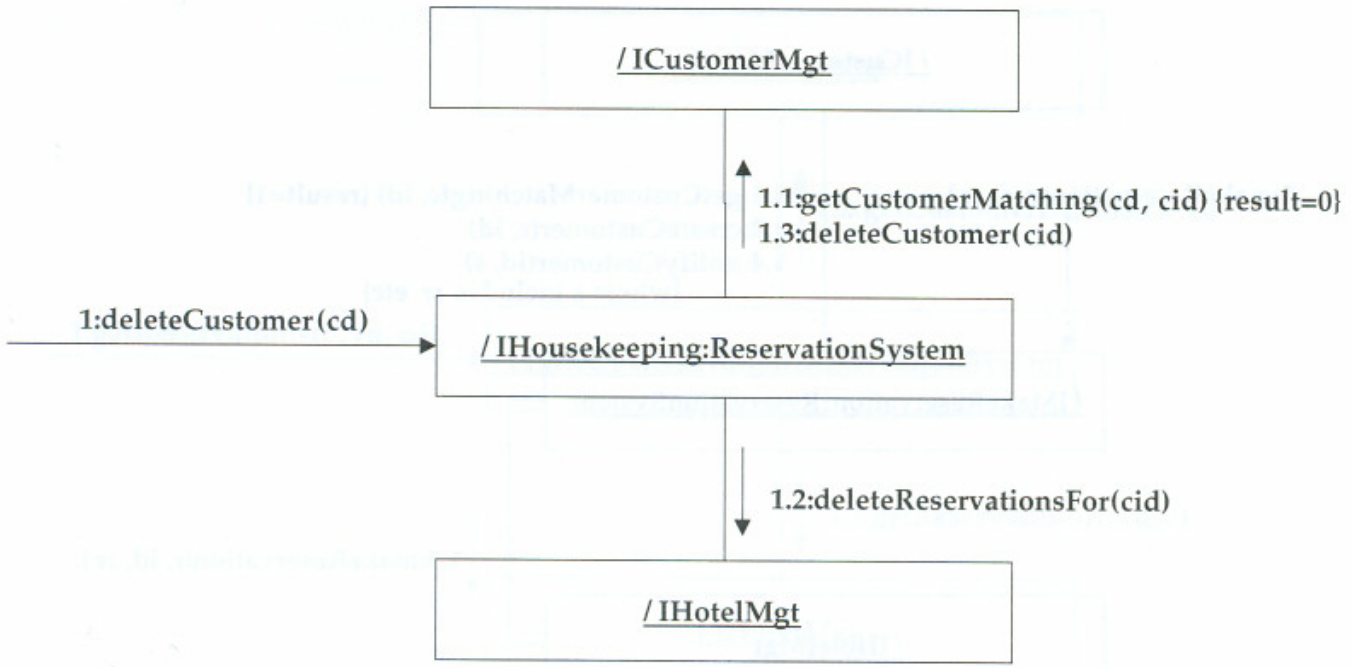


Figure 6.10 Interaction for referential integrity option 3

Disadvantage of option 3: assumes that the CustomerMgr component is object is exclusive to the ReservationSystem.

If this assumption cannot be made, option 2 must be used. Realization using Observer design pattern.

Completing the picture

- What happens if a *new* customer makes a reservation?
- Need for an operation on ICustomerMgt to create a new customer.

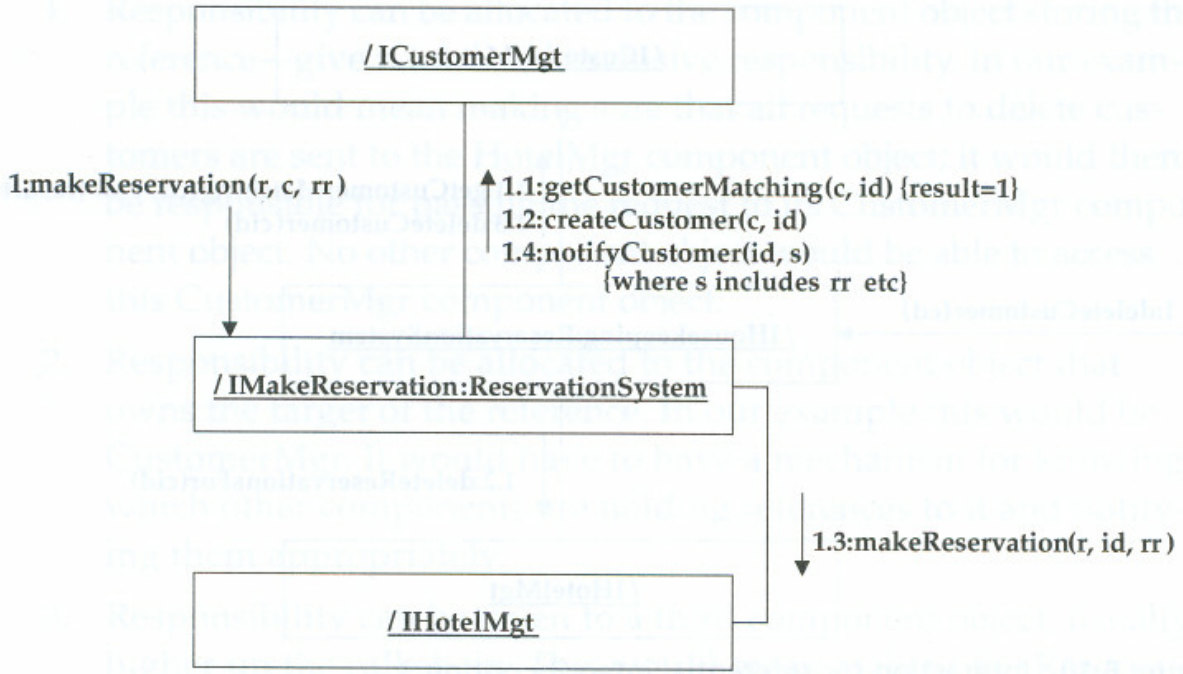


Figure 6.11 makeReservation() interaction (new customer)

Considering the Take Up Reservation Use case also gives rise to new operations on IHotelMgt and ICustomerMgt.

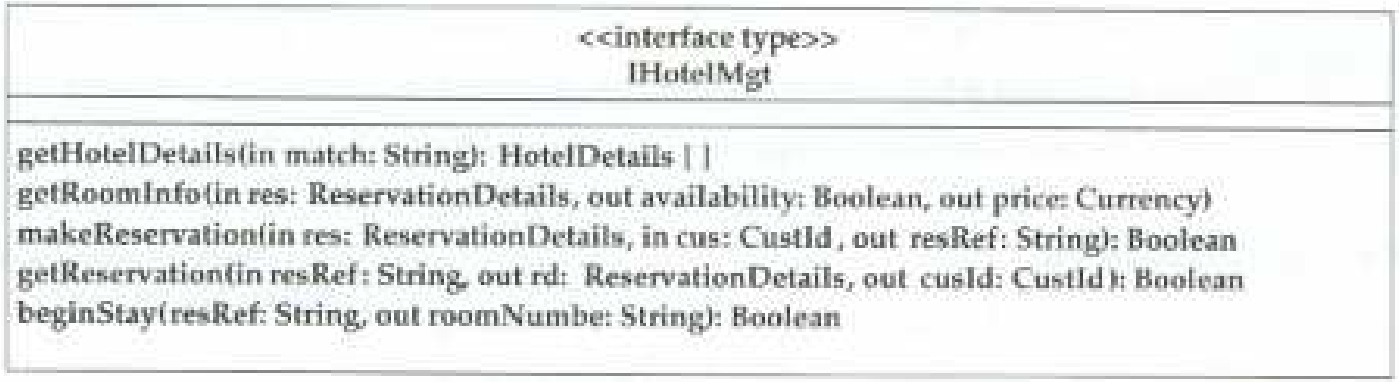
System interfaces with operation signatures

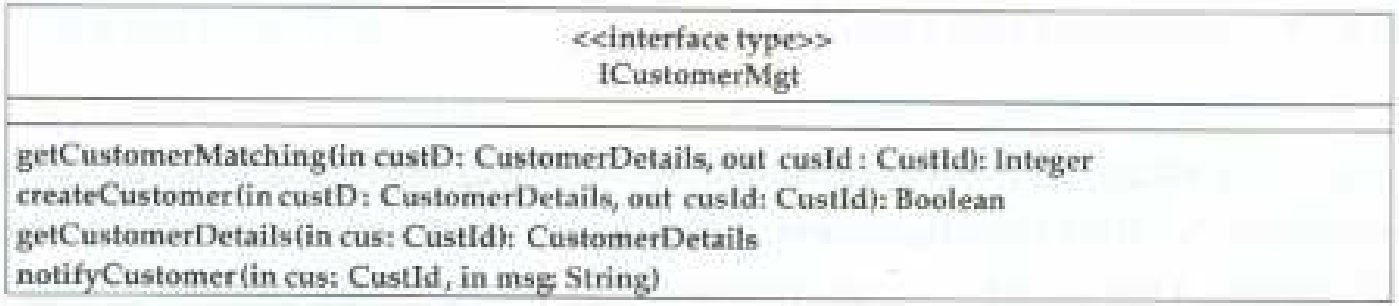
<<interface type>> IMakeReservation
getHotelDetails(in match: String): HotelDetails [ ] getRoomInfo(in res : ReservationDetails, out availability: Boolean, out price: Currency) makeReservation(in res : ReservationDetails, in cus: CustomerDetails, out resRef: String): Integer

<<interface type>> ITakeUpReservation
getReservation(in resRef: String, out rd: ReservationDetails, out cus: CustomerDetails): Boolean beginStay(in resRef: String, out roomNumber: String): Boolean

<<interface type>> IBilling
openAccount(in res : ReservationDetails, in cus: CustomerDetails)

Business interfaces with operation signatures



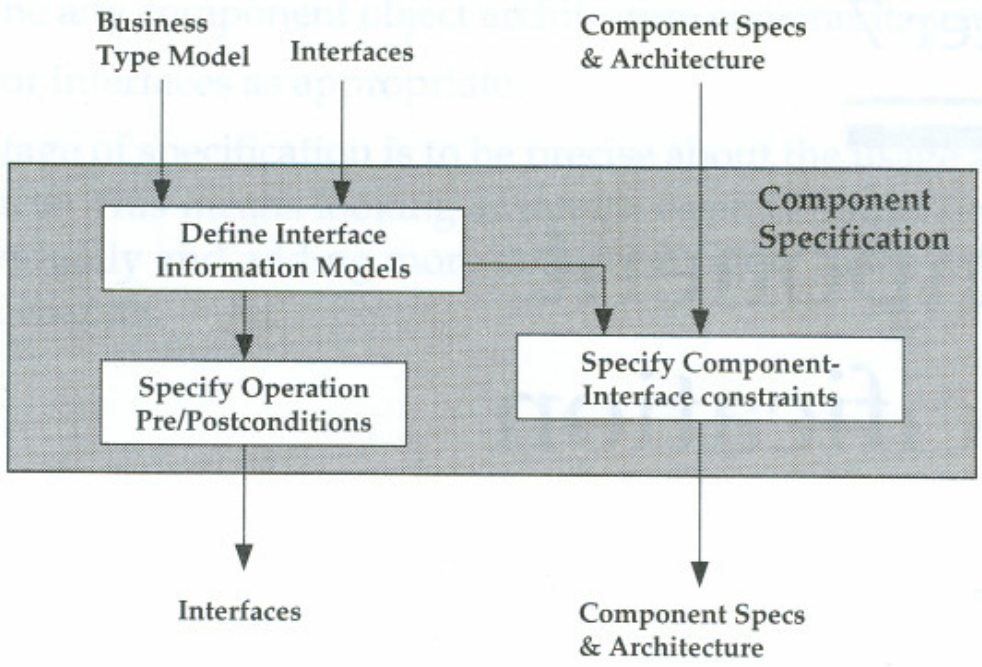


Summary of component interaction

- Develop interaction models for each system interface operation.
- Discover business interface operations and their signatures.
- Refine responsibilities.
- Define any component architecture constraints you need.

Component specification

- A usage contract is defined by an interface specification.
- A realization contract is defined by a component specification.
- Component specifications are primarily groupings of interfaces.
- Component (and interface) specification is the final stage of the specification workflow.



Specifying interfaces

- An interface is a set of operations.
- An operation represents a fine-grained contract between a client and a component object.
- To express the contract, we need a construct that describes the state of a component object.

Operation specification

An operation specifies the individual action that a component object will perform for a client. This has a number of facets:

- The input parameters: specifying the information provided or passed to the component object.
- The output parameters: specifying the information updated or returned by the component object.
- Any resulting change of state of the component object.
- Any constraints that apply (precondition).

However, operation specifications on interfaces do not include information about interactions between the component object performing the operation and other component objects that are required, in a specific implementation, to complete the operation.

Interface information models I

- We need to represent the state of the component on which the interface depends.
- To do this, each interface has an interface information model.
- All changes to the state of the component object caused by a given operation can be described in terms of this information model definition.

Interface information models II

Example:

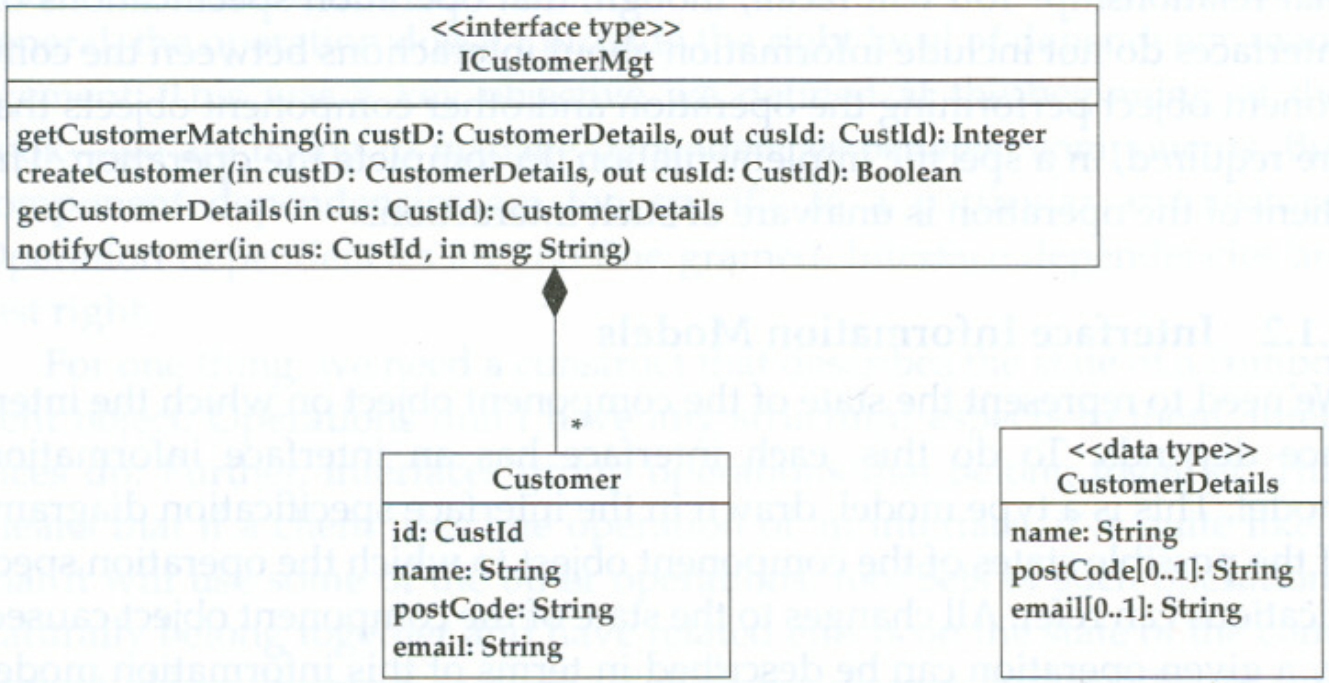


Figure 7.2 Interface specification diagram for the ICustomerMgt interface

Pre- and postconditions I

- Each operation has a pre- and a postcondition.
- These can be defined precisely using OCL.
- The OCL expressions can refer to the operation parameters, the operation result, and the state of the component object (as defined by the interface information model).
- The OCL expressions cannot refer to anything else.

Pre- and postconditions II

Example:

```
context ICustomerMgt::getCustomerDetails
          (cus: CustId): CustomerDetails

pre: -- cus is a valid customer
customer->exists(c: Customer | c.id = cus)

post:
-- the details returned match the details
-- of the customer whose id is cus
-- find the customer

let theCust: Customer = customer->
select(c: Customer| c.id = cus)->asSequence()->first() in
    result.name = theCust.name and
    result.postCode = theCust.postCode and
    result.email = theCust.email
```

From business type model to interface information model I

Result of component identification phase:

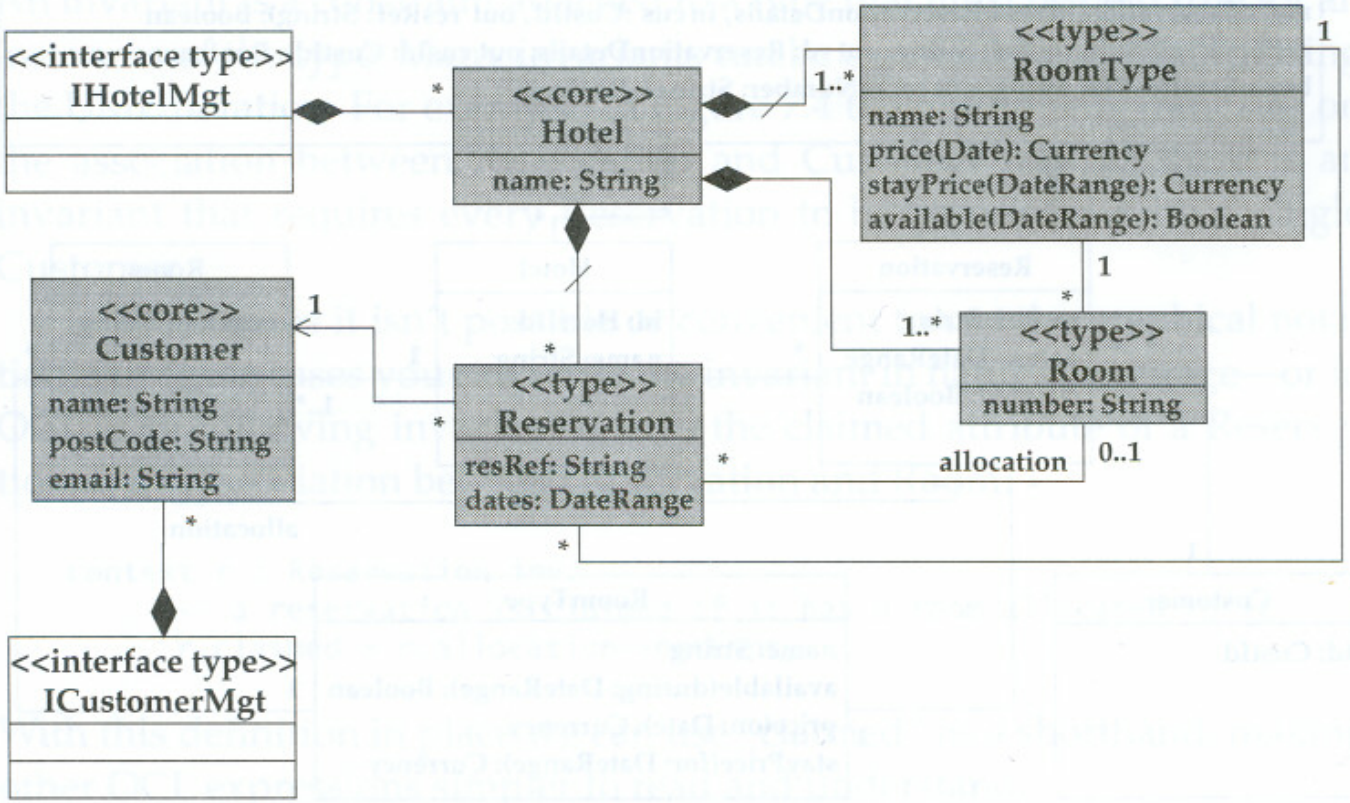


Figure 7.3 Case study interface responsibility diagram

From business type model to interface information model II

- Start by making a copy of the business type model in the interface's package.
- Delete types, associations, and attributes that are not needed.
- When a type owned by one interface refers to a type owned by another, the referenced type (Customer, in this case) appears in the interface information models of both interfaces.
- However, it need not look the same in both interfaces.
- For example, the Customer type in IHotelMgt only needs the customer id.

From business type model to interface information model III

Result:

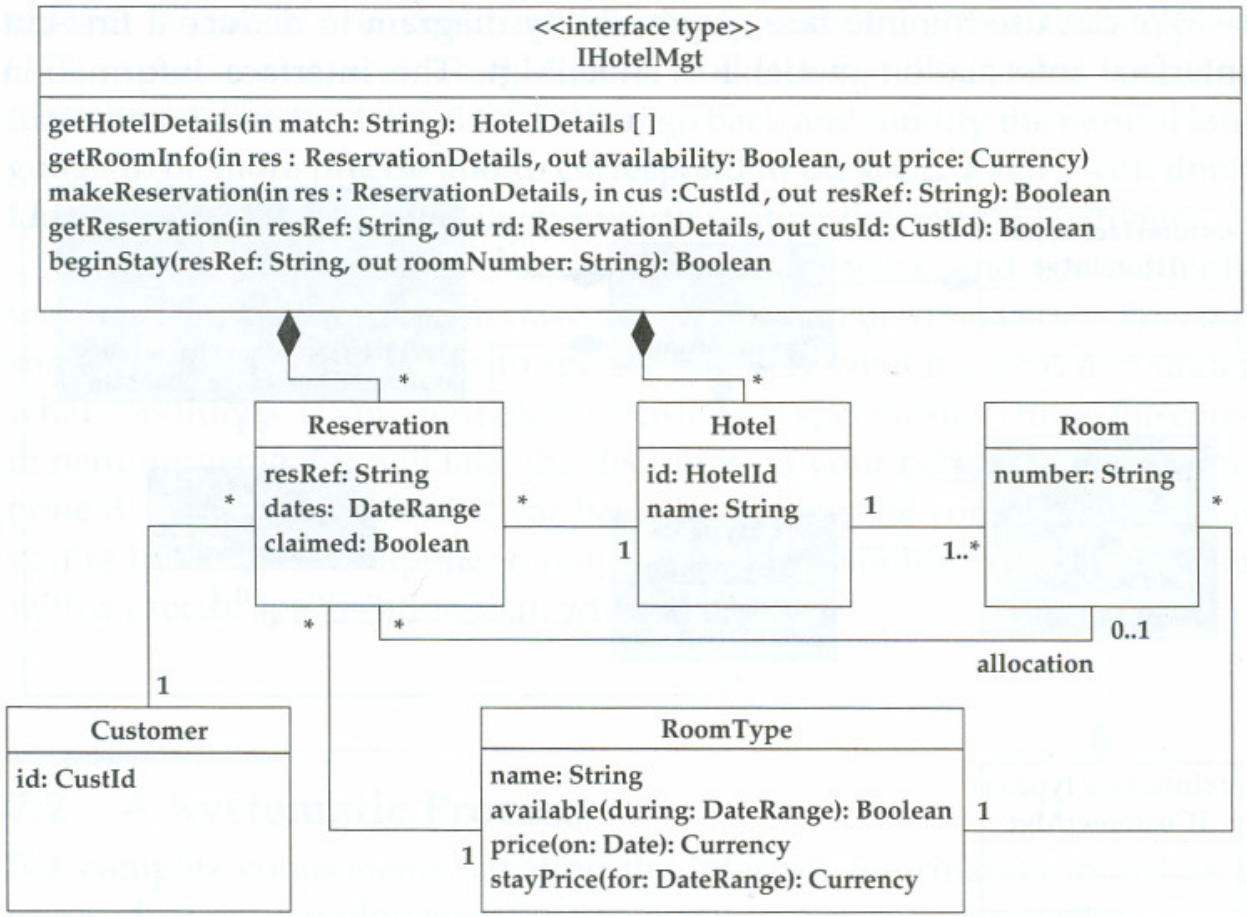


Figure 7.4 Interface specification diagram for IHotelMgt

Invariants

- An invariant is a constraint attached to a type that must be held true for all instances of the type.
- Many invariants can be expressed graphically, using UML notation (e.g., multiplicities).
- In some cases it isn't possible or convenient to use the graphical notation. Use OCL instead.

Example:

```
context Reservation
-- a reservation is claimed
-- if it has a room allocated to it
inv:  claimed = allocation->notEmpty()
```

Snapshots

A useful technique when writing pre- and postconditions is to draw "before" and "after" instance diagrams and to highlight the state changes that occur.

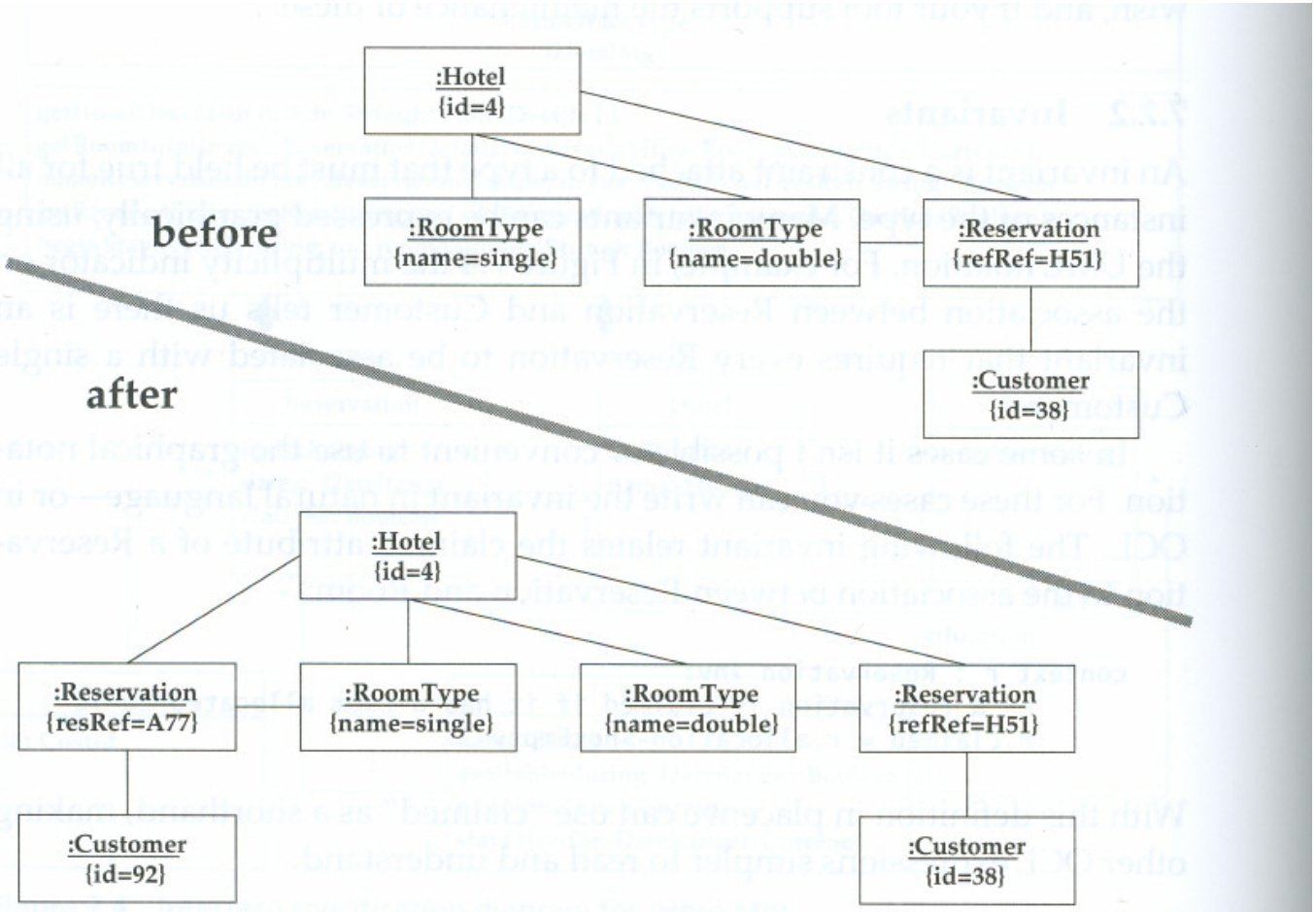


Figure 7.5 "Before" and "after" snapshot instance diagrams for IHotelMgt::makeReservation()

Specification of IHotelMgt::makeReservation I


```
context IHotelMgt::makeReservation
   (res: ReservationDetails, cus: CustId, resRef: String)
   : Boolean

pre:
-- the hotel id and room type are valid
hotel->exists(h | h.id = res.hotel
      and h.room.roomType.name->includes(res.roomType))

post:
result implies
-- a reservation was created
-- identify the hotel
```

Specification of IHotelMgt::makeReservation II


```
let h: Hotel = hotel->select(x | x.id = res.hotel)
->asSequence()->first() in

-- only one more reservation now than before
h.reservation->size() - h.reservation@pre->size() = 1

-- identify the reservation
and let r: Reservation = h.reservation->
    select(y: Reservation| not h.reservation@pre->
        includes(y))->asSequence()->first() in

-- return number is number of the new reservation
r.resRef = resRef and
```

Specification of IHotelMgt::makeReservation III


```
-- other attributes match
r.dates = res.dates and
r.roomType.name = res.roomType
and not r.claimed and
r.customer.id = cus
```

Specifying system interfaces I

- Until now, we have discussed a systematic way of moving from the business type model to the information models of the business interfaces.
- For the system interfaces, we take a similar approach.
- As with any other interface, the interface information model of a system interface needs to contain just enough information for the operations to be specified.
- This will be a subset of the business type model.
- Note that the existence of an interface information model does not imply that an implementation of the interface must store the information persistently. In fact, system interfaces rarely have persistent storage.

Specifying system interfaces II

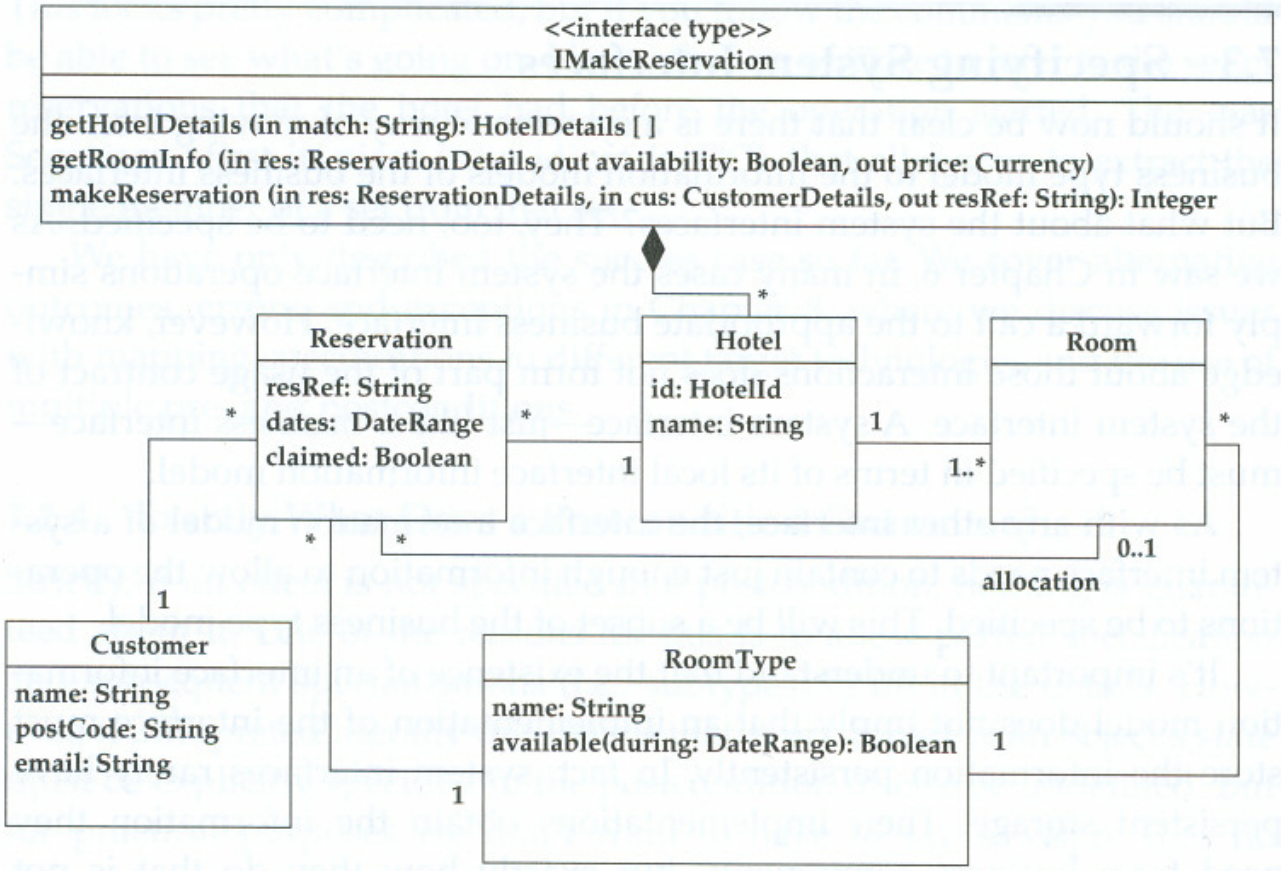


Figure 7.6 Interface specification diagram for IMakeReservation

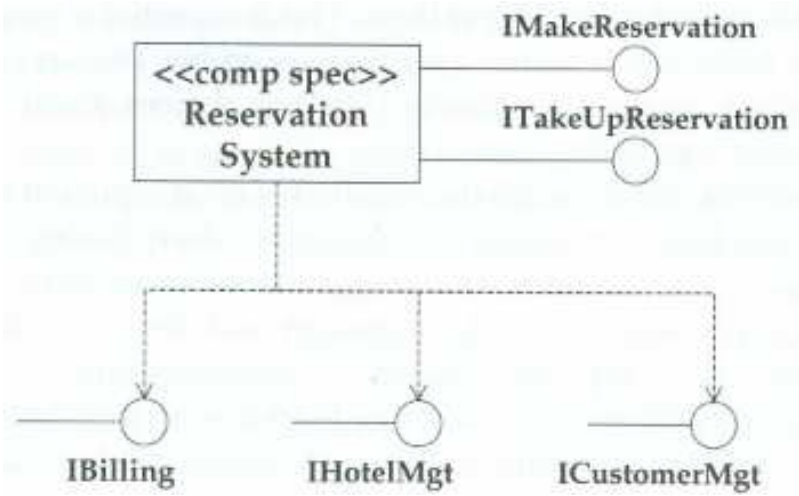
Note that the information model for IMakeReservation does not require the room number attribute, so it has been removed.

Specifying components

- The interface specifications discussed so far deal with the usage contract – the contract between a component object and its clients.
- Now we consider the additional specification information that the component implementer and assembler need to be aware of, especially the dependencies of a component on other interfaces.
- This information forms the component specification.

Offered and used interfaces

- For every component specification we need to say which interfaces its realizations must support, see architecture diagram of component identification phase.
- Now, we must dissect that diagram into pieces specific to each component specification.
- We also need to confirm any constraints concerning which other interfaces are to be used by a realization (dependency arrows in architecture diagram).



Scoping interactions I

- Constraints how a particular operation must be implemented are defined in interactions.
- Component interactions define specification-level constraints. All component realizations must respect them.
- This is essential if we aim to be able to replace components within a complex component assembly.
- The interactions that make up the constraints on component specifications are typically fragments of the interactions we drew during operation discovery.
- They begin with a component object receiving a message, and only show the direct interactions from that component.

Scoping interactions II

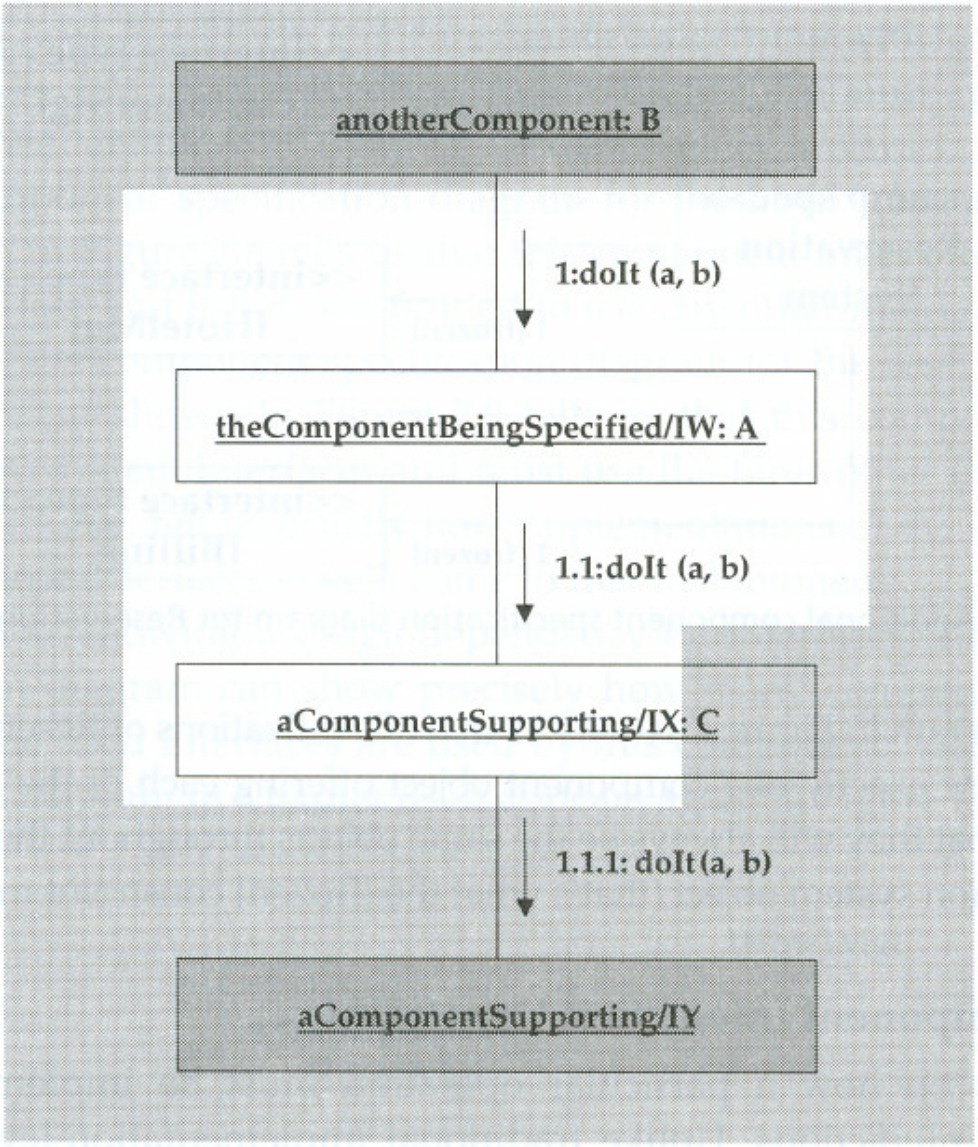


Figure 7.11 Scoping an interaction

Inter-interface constraints

We may want to express constraints concerning the relationships between interface information models. This concerns

- how offered interfaces relate to each other
- how offered interfaces relate to used interfaces.

Offered interfaces

- The Reservation System component offers the IMakeReservation and ITakeUpReservation interfaces.
- Both these interfaces have a Reservation information type.
- Since the two interfaces are specified completely independently, we cannot assume that that both reservation types are the same.
- This has to be expressed explicitly.

```
context ReservationSystem
-- constraints between offered interfaces
IMakeReservation::hotel = ITakeUpReservation::hotel
IMakeReservation::reservation =
                    ITakeUpReservation::reservation
IMakeReservation::customer = ITakeUpReservation::customer
```

where a formal definition of "=" depends on the two information types involved.

Offered and used interfaces

- Note that the existence of an interface information model does not imply that implementations of the interface will store the information.
- Instead, they obtain the information from the business components.
- Therefore, we write constraints that require the elements of the interface information models to match up.

```
context ReservationSystem
-- constraints between offered and used interfaces
IMakeReservation::hotel = IHotelMgt::hotel
IMakeReservation::reservation =
                    IHotelMgt::reservation
IMakeReservation::customer = ICustomerMgt::customer
```

Factoring interfaces I

- Each interface has its own interface information model, which is often only slightly different from the model of another interface.
- Sometimes it is possible to simplify things by refactoring the interfaces, especially by introducing new abstract interfaces that act as super-types of other interfaces, holding common interface information model elements, and, sometimes, definitions of common operations.
- In some cases it may even be practical to simply merge system interfaces together and do not bother with subtyping. This may be appropriate when the corresponding use cases have the same actors.

Factoring interfaces II

Example: factor out the common elements of the information models from IMakeReservation and ITakeUpReservation and place them in a new interface called IReservationSystem:

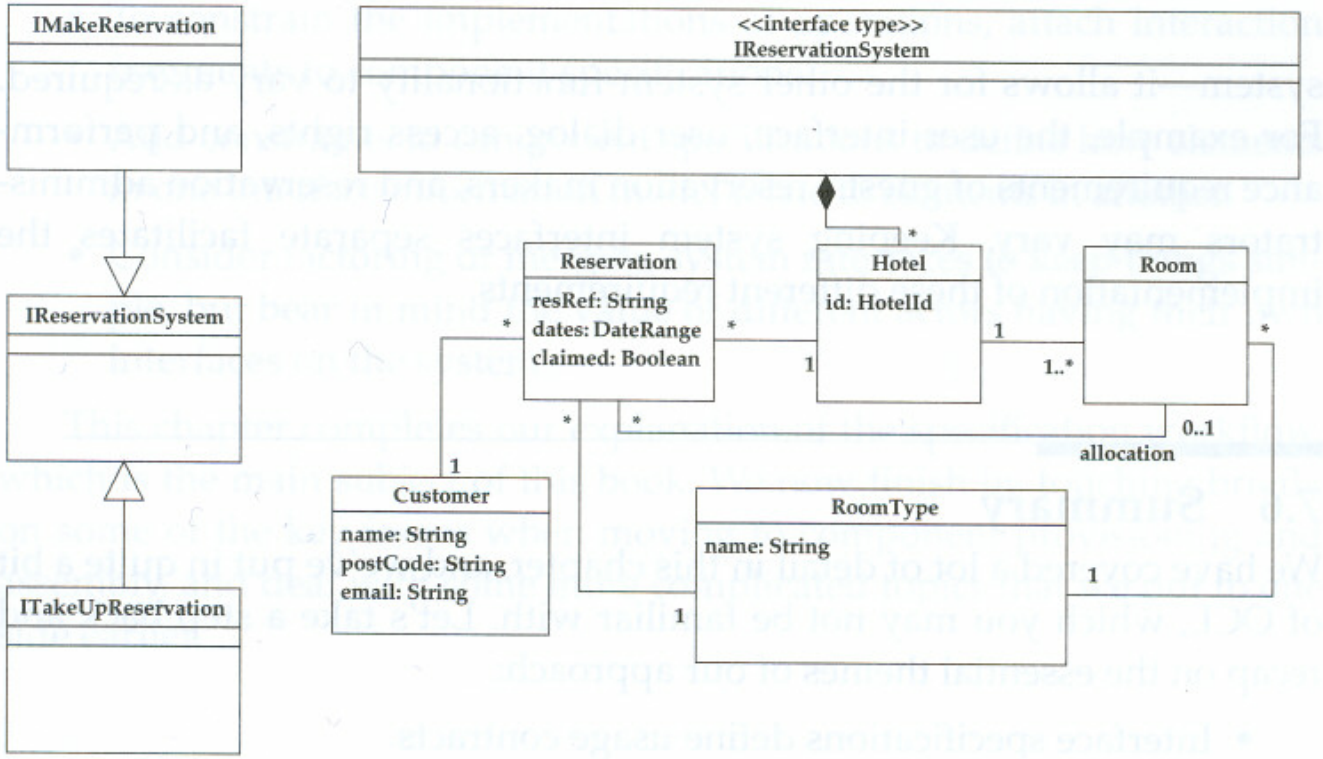


Figure 7.12 Refactoring interfaces

Factoring interfaces III

The interface information model for IMakeReservation then merely extends the inherited types, adding extra attributes that are required.

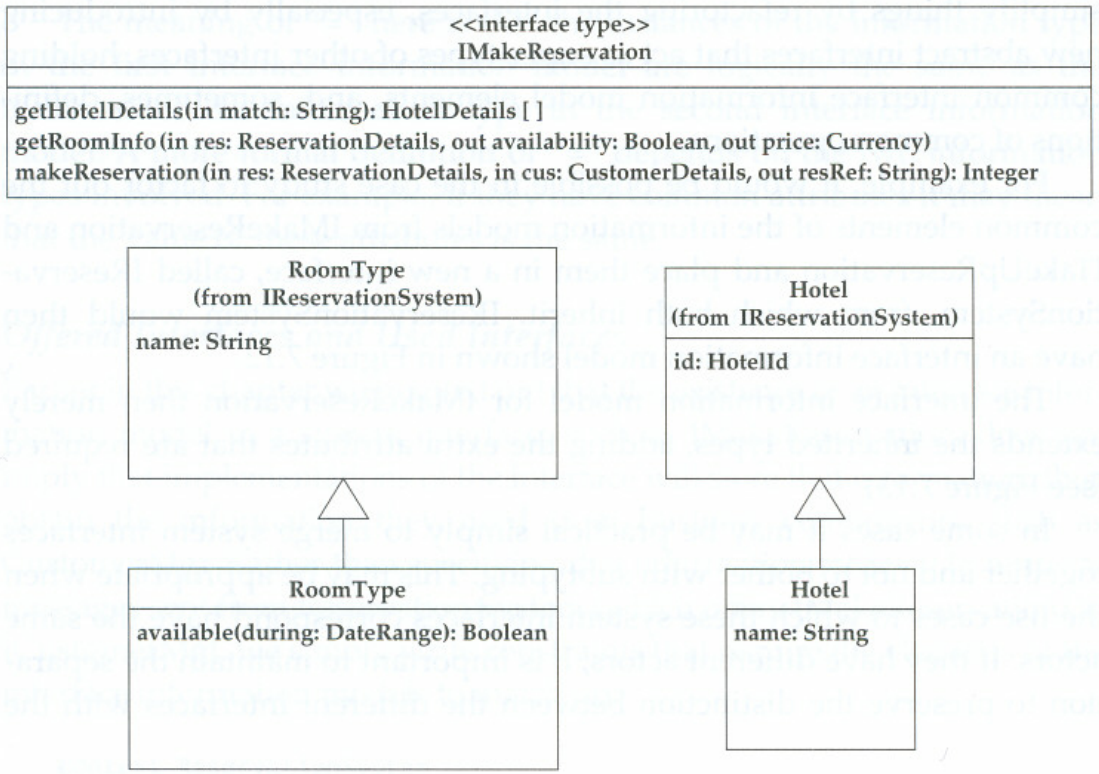


Figure 7.13 IMakeReservation after factoring out IReservationSystem

Summary of component specification I

- Interface specifications define usage contracts.
- Component specifications define realization contracts.
- An interface is specified by a set of operation specifications that operate on an interface information model.
- The interface information model must contain just enough information to allow the operations to be specified. It cannot refer to anything outside the interface.
- First-cut interface information models can be derived systematically from the business type model.
- Each operation is specified using a pre- and postcondition pair.

Summary of component specification II

- OCL can be used to express invariants and operation pre- and postconditions.
- Component specifications include specifications of the interfaces offered and used.
- To constrain the implementations of operations, attach interaction fragments to component specifications.
- Add constraints to component specifications to define how elements in one interface information model relate to elements in another.
- Consider factoring or merging system interfaces to keep things simple, but bear in mind the value of different actors having their own interfaces on the system.

Provisioning and Assembly

- In the specification workflow, we have been working in a technology-independent way.
- Provisioning means to provide component implementations, either by directly implementing the specifications or by finding an existing component that fits the specification.
- Assembly pulls the components together, using the component architecture for the software to define the overall structure and the individual pieces, and adding user interface and dialog logic.

Issues in provisioning

- A component realizes a component specification and an interface realizes an interface type.
- The realizations are performed in some target technology.
- We must consider what mappings need to take place for these two key realizations, between the technology-neutral and the technology-specific level.
- Main issues:
  - Operation parameter type, kind (in/out/inout/return), and reference restrictions
  - Exception and error handling mechanisms (implementing the contracts)
  - Interface inheritance and support restrictions
  - Operation sequence
  - Interface properties
  - Object creation mechanisms
  - Raising events

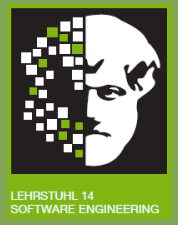
Bibliography I

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley, Boston, MA, USA, 1st edition.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

Cheesman, J. and Daniels, J. (2001). *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley.

Coplien, J. O. (1992). *Advanced C++ Programming Styles and Idioms*. Addison-Wesley.



Bibliography II

Coplien, J. O. (1998). C++ idioms. `http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html` (last visit: May 27th, 2009).

D'Souza, D. and Wills, A. C. (1998). *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading.

Heineman, G. T. and Councill, W. T. (2001). *Component-Based Software Engineering*. Addison-Wesley.

Bibliography III

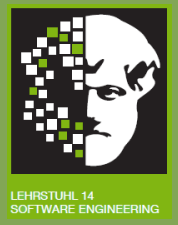
Heisel, M., Santen, T., and Souquières, J. (2002). Toward a formal model of software components. In *Proc. 4th International Conference on Formal Engineering Methods*, pages 57–68. Springer.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall International, 2nd edition.

OSGi Alliance (2010a). *OSGi Service Platform Release 4 Version 4.2 Compendium Specification*. `http://www.osgi.org/Download/Release4V42`.

OSGi Alliance (2010b). *OSGi Service Platform Release 4 Version 4.2 Core Specification*. `http://www.osgi.org/Download/Release4V42`.

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software*. Pearson Education. Second edition.



Bibliography IV

Wütherich, G., Hartmann, N., Kolb, B., and Lübken, M. (2008). *Die OSGi Service Platform: Eine Enführung mit Eclipse*. dpunkt.