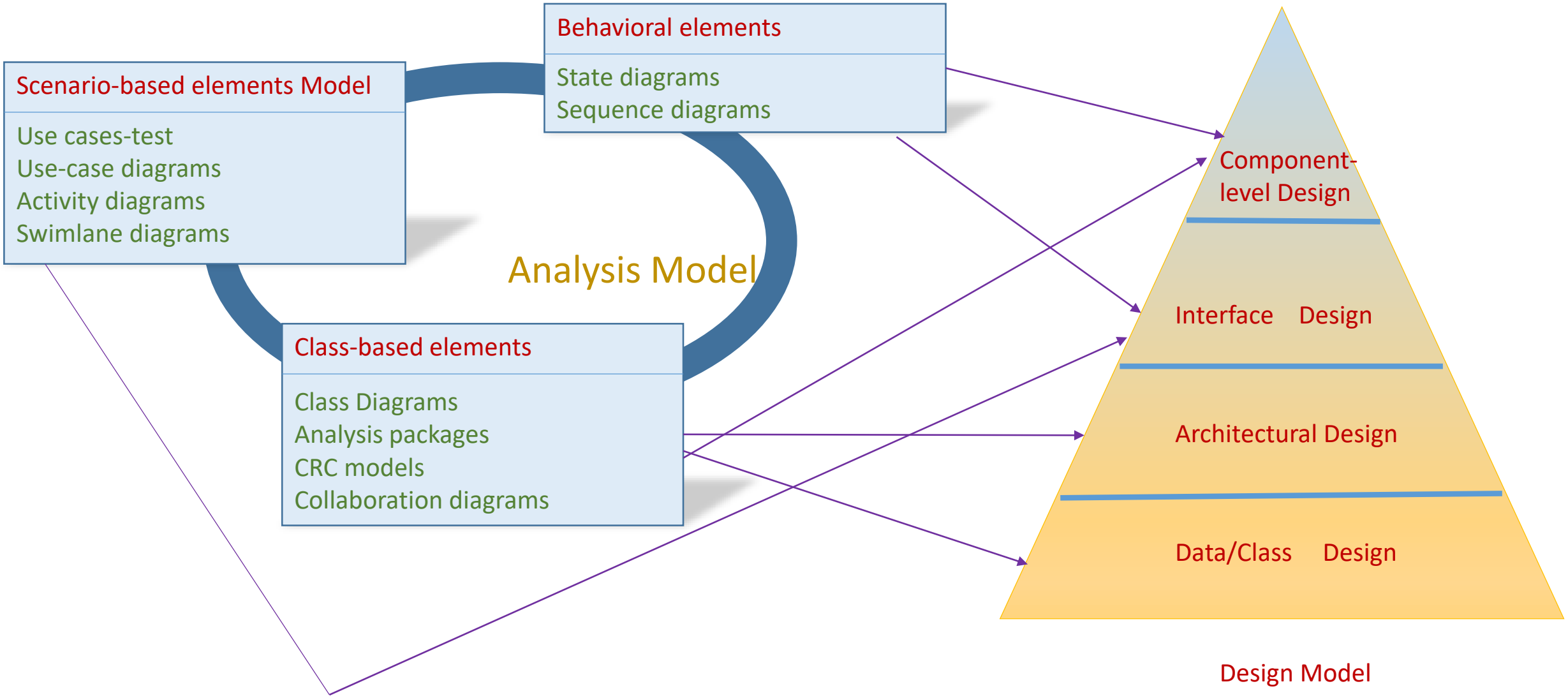
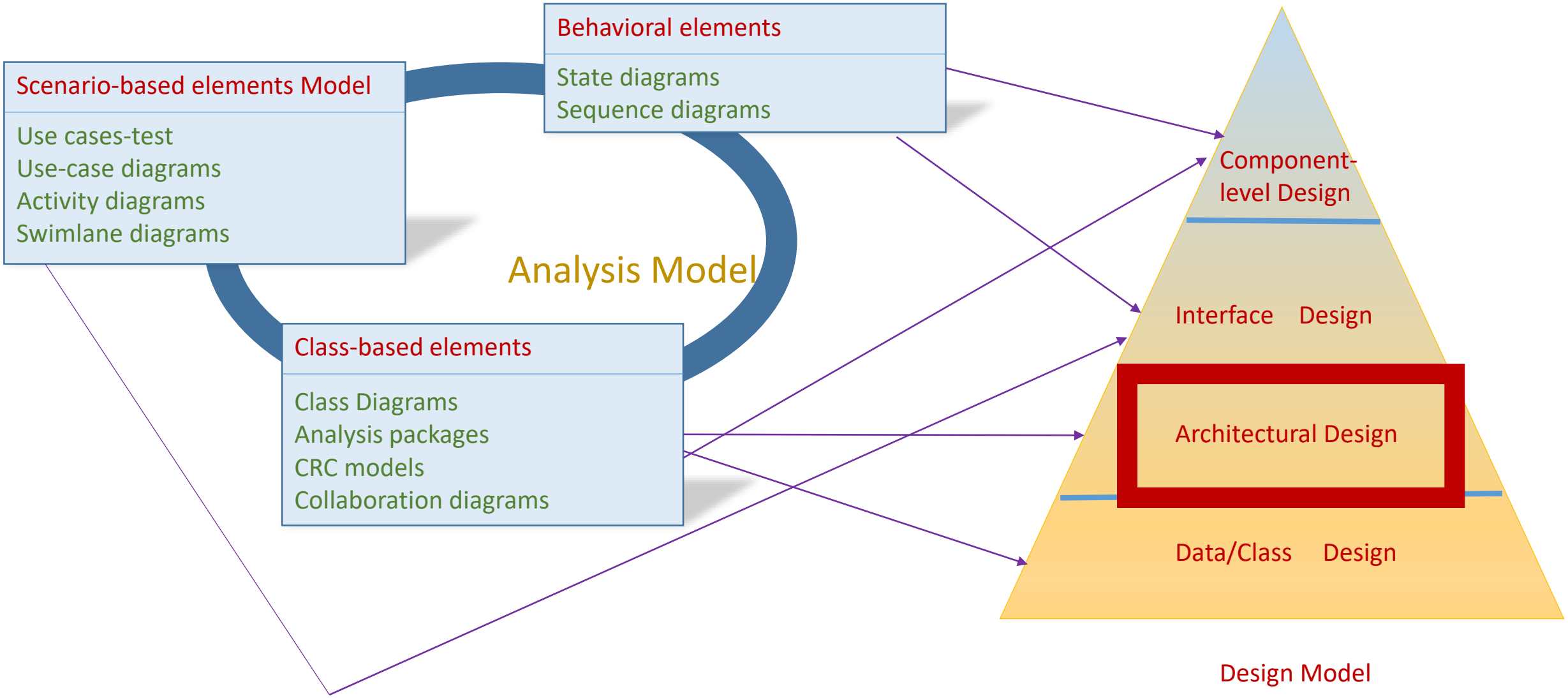
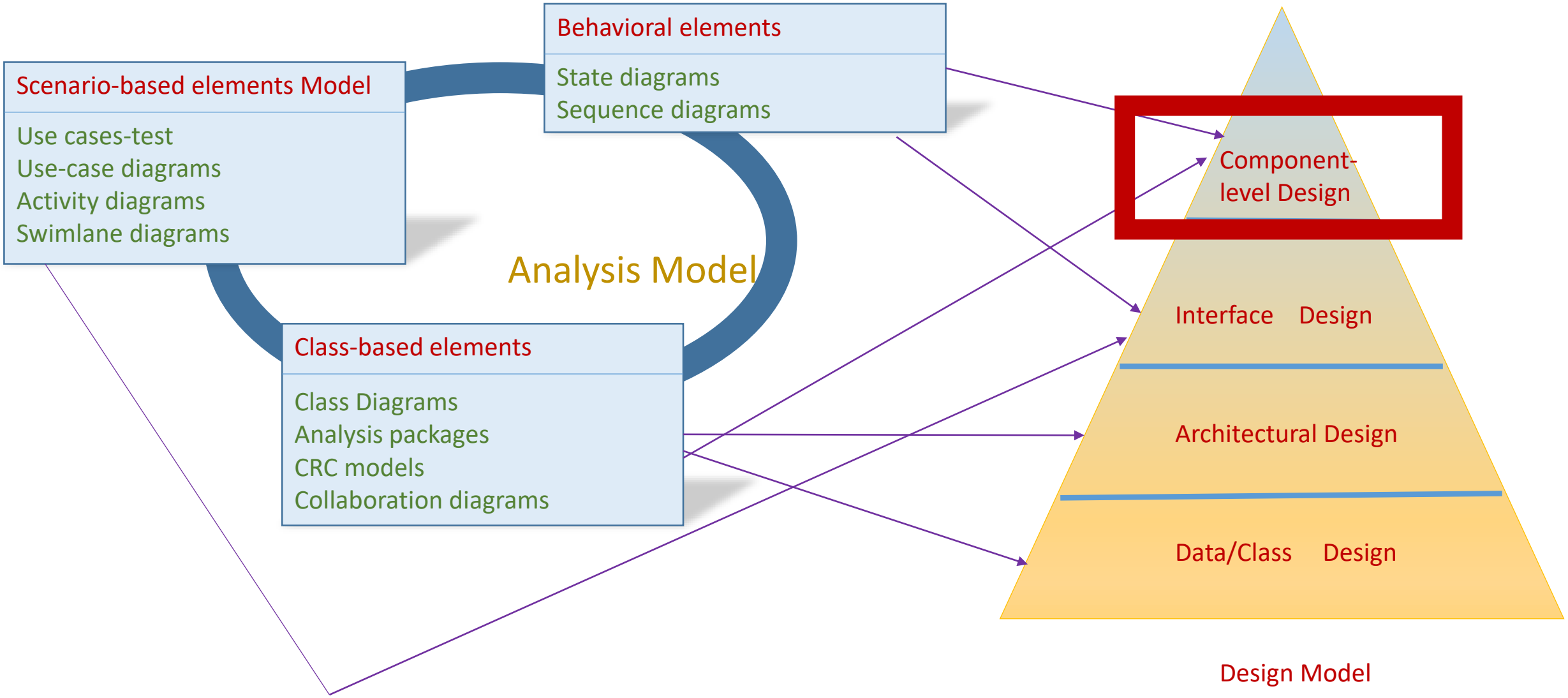
Design II

Jingshu Chen


Behavioral elements
State diagrams
Sequence diagrams
Scenario-based elements Model
Use cases-test
Use-case diagrams
Activity diagrams
Swimlane diagrams
Analysis Model
Class-based elements
Class Diagrams
Analysis packages
CRC models
Collaboration diagrams
Component-level Design
Interface Design
Architectural Design
Data/Class Design
Design Model


Behavioral elements
State diagrams
Sequence diagrams
Scenario-based elements Model
Use cases-test
Use-case diagrams
Activity diagrams
Swimlane diagrams
Analysis Model
Class-based elements
Class Diagrams
Analysis packages
CRC models
Collaboration diagrams
Component-level Design
Interface Design
Architectural Design
Data/Class Design
Design Model


Behavioral elements
State diagrams
Sequence diagrams
Scenario-based elements Model
Use cases-test
Use-case diagrams
Activity diagrams
Swimlane diagrams
Analysis Model
Class-based elements
Class Diagrams
Analysis packages
CRC models
Collaboration diagrams
Component-level Design
Interface Design
Architectural Design
Data/Class Design
Design Model

Component-level Design

- What is a component?

Component-level Design

- What is a component?

"A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

--the OMG unified Modeling Language specification

Component-level Design

- Architectural design: a complete set of software components

- Component-level design:
  - The data structure, algorithms, interface characteristics, and communication mechanism allocated to each component

Component-level Design

- Basic Design Principles

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

  A module should be open for extension but closed for modification.

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

  A module should be open for extension but closed for modification.

  Example:

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

Component-level Design

- The Open-Closed Principles(OCP)

  A module should be open for extension but closed for modification.

  Example:

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Component-level Design

- The Open-Closed Principles(OCP)

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

n.

Component-level Design

- The Open-Closed Principles(OCP)

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

How about triangles?

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

A module should be open for extension but closed for modification.

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

A module should be open for extension but closed for modification.

```
public abstract class Shape
{
    public abstract double Area();
}
```

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

A module should be open for extension but closed for modification.

```
public abstract class Shape
{
    public abstract double Area();
}
```

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

n.

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)

A module should be open for extension but closed for modification.

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Basic Design Principles
  - The Liskov Substitution Principle(LSP)

  Subclasses should be substitutable for their base classes.

Component-level Design

- The Liskov Substitution Principle(LSP)

Component-level Design

- The Liskov Substitution Principle(LSP)

```
// Violation of Likov's Substitution Principle
class Rectangle
{
        protected int m_width;
        protected int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int height){
                m_height = height;
        }


        public int getWidth(){
                return m_width;
        }

        public int getHeight(){
                return m_height;
        }

        public int getArea(){
                return m_width * m_height;
        }
}
```

```
class Square extends Rectangle
{
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }

}
```

Component-level Design

- The Liskov Substitution Principle(LSP)

```
class LspTest
{
        private static Rectangle getNewRectangle()
        {
                // it can be an object returned by some factory ...
                return new Square();
        }

        public static void main (String args[])
        {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle.
                // It assumes that he's able to set the width and height as for the base class

                System.out.println(r.getArea());
                // now he's surprised to see that the area is 100 instead of 50.
        }
}
```

Component-level Design

- The Liskov Substitution Principle(LSP)

```
class LspTest
{
        private static Rectangle getNewRectangle()
        {
                // it can be an object returned by some factory ...
                return new Square();
        }

        public static void main (String args[])
        {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle.
                // It assumes that he's able to set the width and height as for the base class

                System.out.println(r.getArea());
                // now he's surprised to see that the area is 100 instead of 50.
        }
}
```

Make sure the new derived classes are extending the base classes without changing their behavior!

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on concretions.

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on concretions.

```
class Worker {

        public void work() {

                // ....working

        }

}
```

```
class Manager {

        Worker worker;



        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}

class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on conc

```
class Worker {

        public void work() {

                // ....working

        }

}
```

```
class Manager {

        Worker worker;



        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}

class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on concretions.

```
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}

class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }
}
```

```
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on concre

```
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}

class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }
}
```

```
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- Dependency Inversion Principle(DIP)

Depend on abstractions. Do not depend on concre

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one general purpose interface.

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one general purpose interface.

```
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ..... eating in launch break
        }
}
```

```
class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than or
interface.

```
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ..... eating in launch break
        }
}
```

```
class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one general purpose interface.

```
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
        public void work();
}

interface IFeedable{
        public void eat();
}

class Worker implements IWorkable, IFeedable{
        public void work() {
                // ....working
        }

        public void eat() {
                //.... eating in launch break
        }
}
```

```
class Robot implements IWorkable{
        public void work() {
                // ....working
        }
}

class SuperWorker implements IWorkable, IFeedable{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        Workable worker;

        public void setWorker(Workable w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one interface.

```
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
        public void work();
}

interface IFeedable{
        public void eat();
}

class Worker implements IWorkable, IFeedable{
        public void work() {
                // ....working
        }

        public void eat() {
                //.... eating in launch break
        }
}
```

```
class Robot implements IWorkable{
        public void work() {
                // ....working
        }
}

class SuperWorker implements IWorkable, IFeedable{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        Workable worker;

        public void setWorker(Workable w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Component-level Design

- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one general purpose interface.

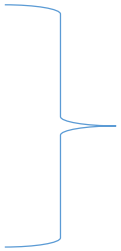
Clients should not be forced to depend upon interfaces that they don't use.

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)



OO Package Design Principles

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)

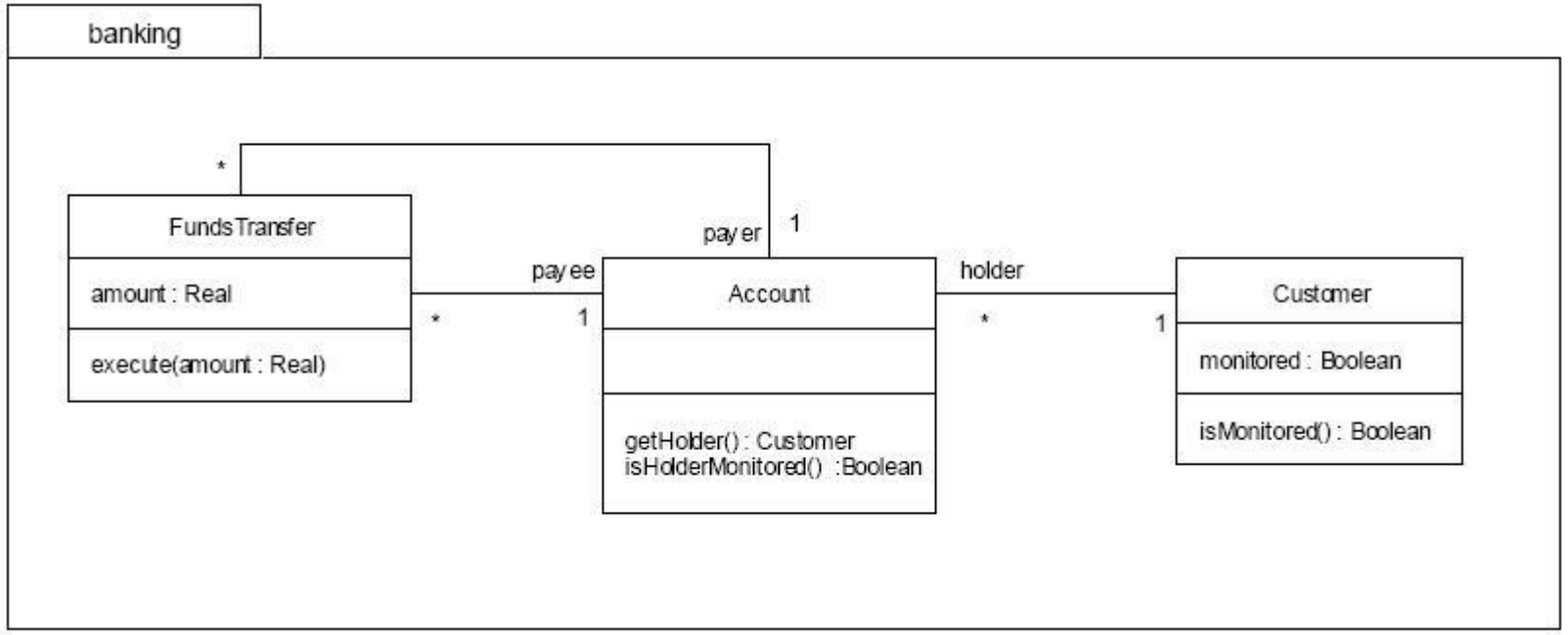
Component-level Design

- The Release Reuse Equivalency Principle(REP)

The granule of reuse is the granule of release

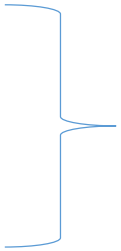
Component-level Design

- The Release Reuse Equivalency Principle(REP)



Component-level Design

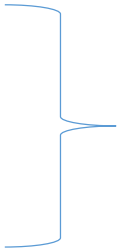
- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)



OO Package Design Principles

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)



OO Package Design Principles

Component-level Design

- The Common closure Principle(CCP)

Classes that change together belong together

Component-level Design

- The Common closure Principle(CCP)

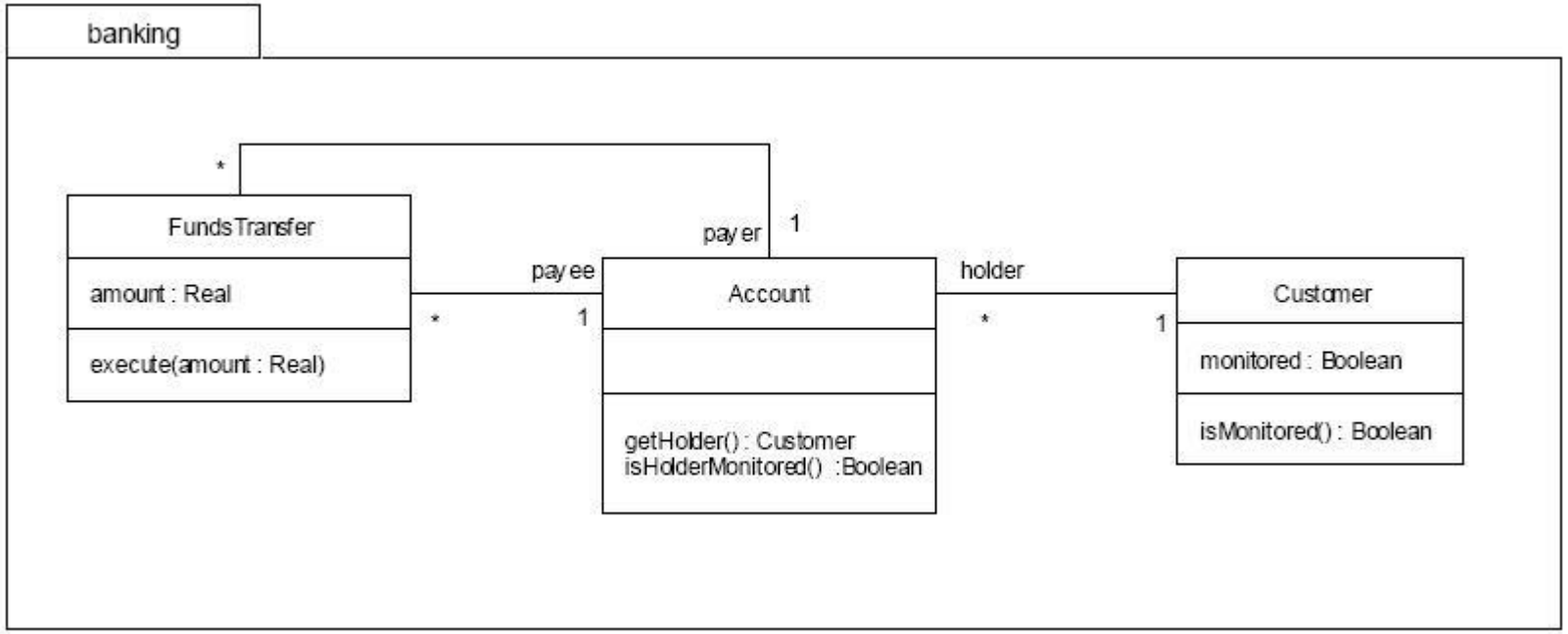
Classes that change together belong together

Maintainability

Component-level Design

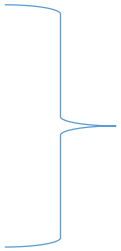
- The Common closure Principle(CCP)

Classes that change together belong together



Component-level Design

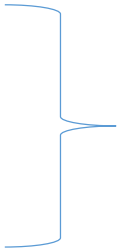
- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)



OO Package Design Principles

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)



OO Package Design Principles

Component-level Design

- The Common Reuse Principle(CRP)

Classes that aren’t reused together should not be grouped together

Component-level Design

- Basic Design Principles
  - The Open-Closed Principles(OCP)
  - The Liskov Substitution Principle(LSP)
  - Dependency Inversion Principle(DIP)
  - The Interface Segregation Principle(ISP)
  - The Release Reuse Equivalency Principle(REP)
  - The Common closure Principle(CCP)
  - The Common Reuse Principle(CRP)
- How to conduct component-level design?

OO Package Design Principles

Conducting Component-level Design

- Step 1. Identify all ***design classes*** that correspond to the problem domain, infrastructure domain.

- Step 2. Elaborate all ***design classes*** (not acquired as reusable components).
- Step 2a. Specify ***message*** details when classes or component collaborate.
- Step 2b. Identify appropriate ***interfaces*** for each component.
- Step 2c. Elaborate ***attributes*** and define data types and data structures.
- Step 2d. Describe processing flow within each ***operation*** in detail.

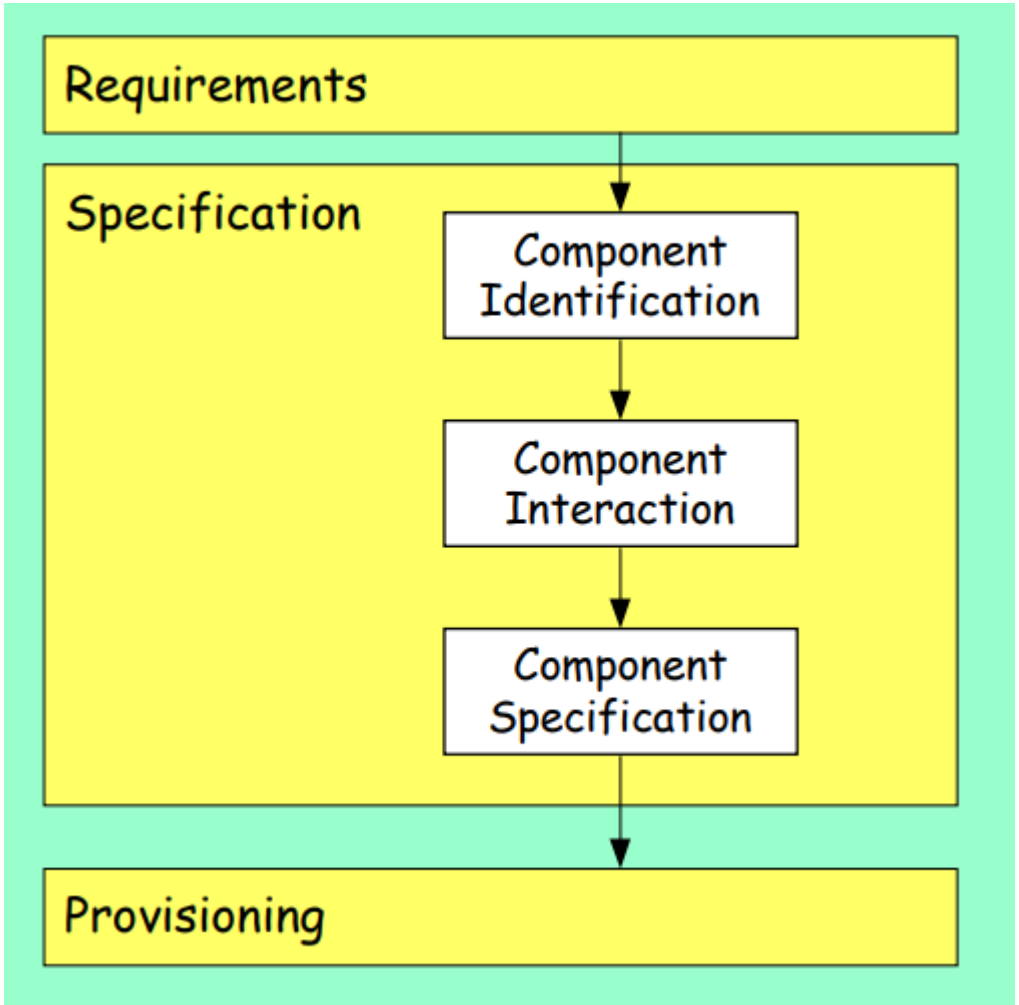
- Step 3. Describe ***persistent*** data sources (databases and files) and identify the ***classes*** required to manage them.
- Step 4. Develop and elaborate ***behavioral*** representations for a class or component.
- Step 5. Elaborate ***deployment*** diagrams to provide additional implementation detail.
- Step 6. Factor every component-level design representation and always consider alternatives.

TOOLS Europe 2000

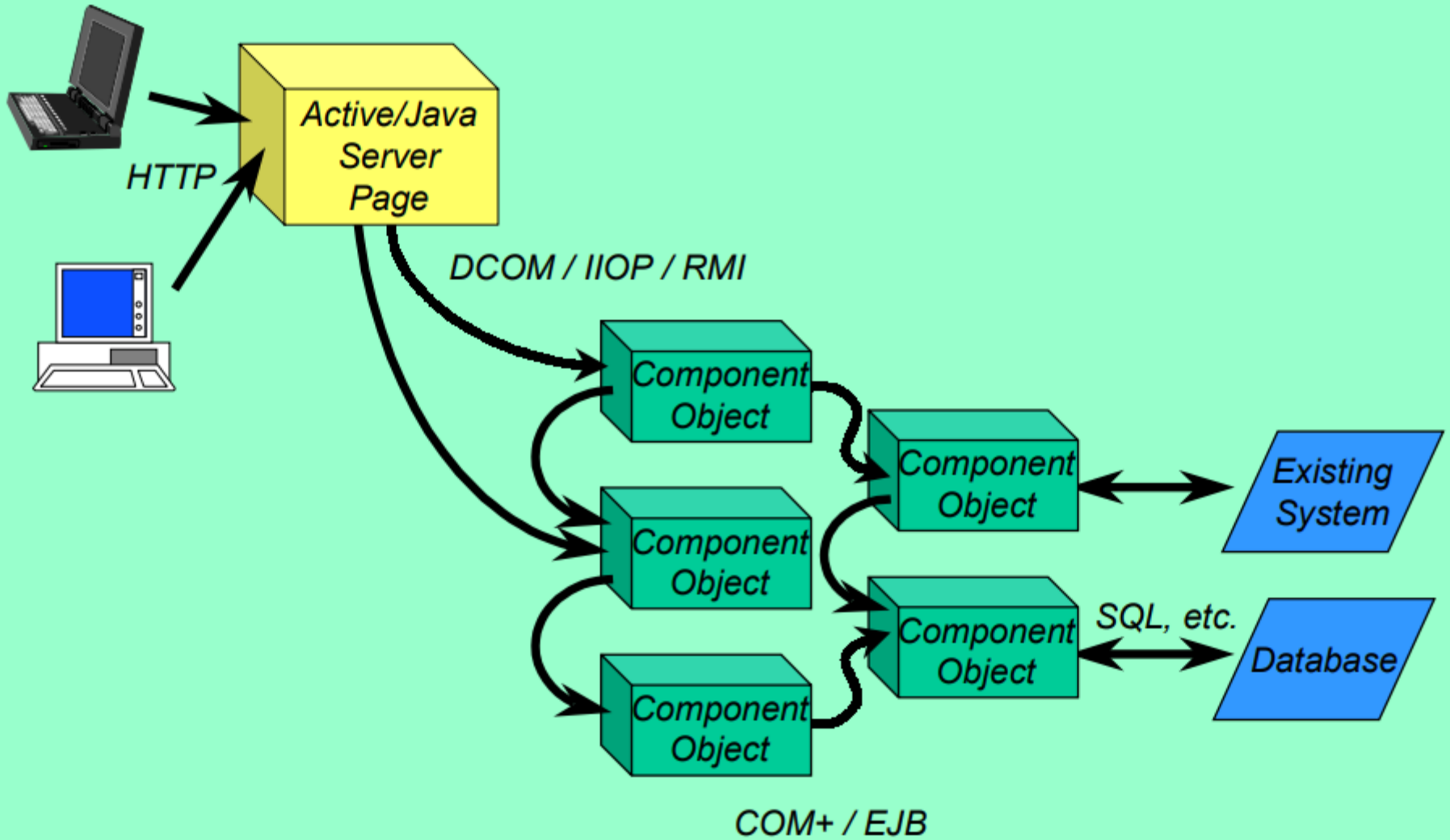
Component-Based Design: A Complete Worked Example

John Daniels
Syntropy Ltd, UK

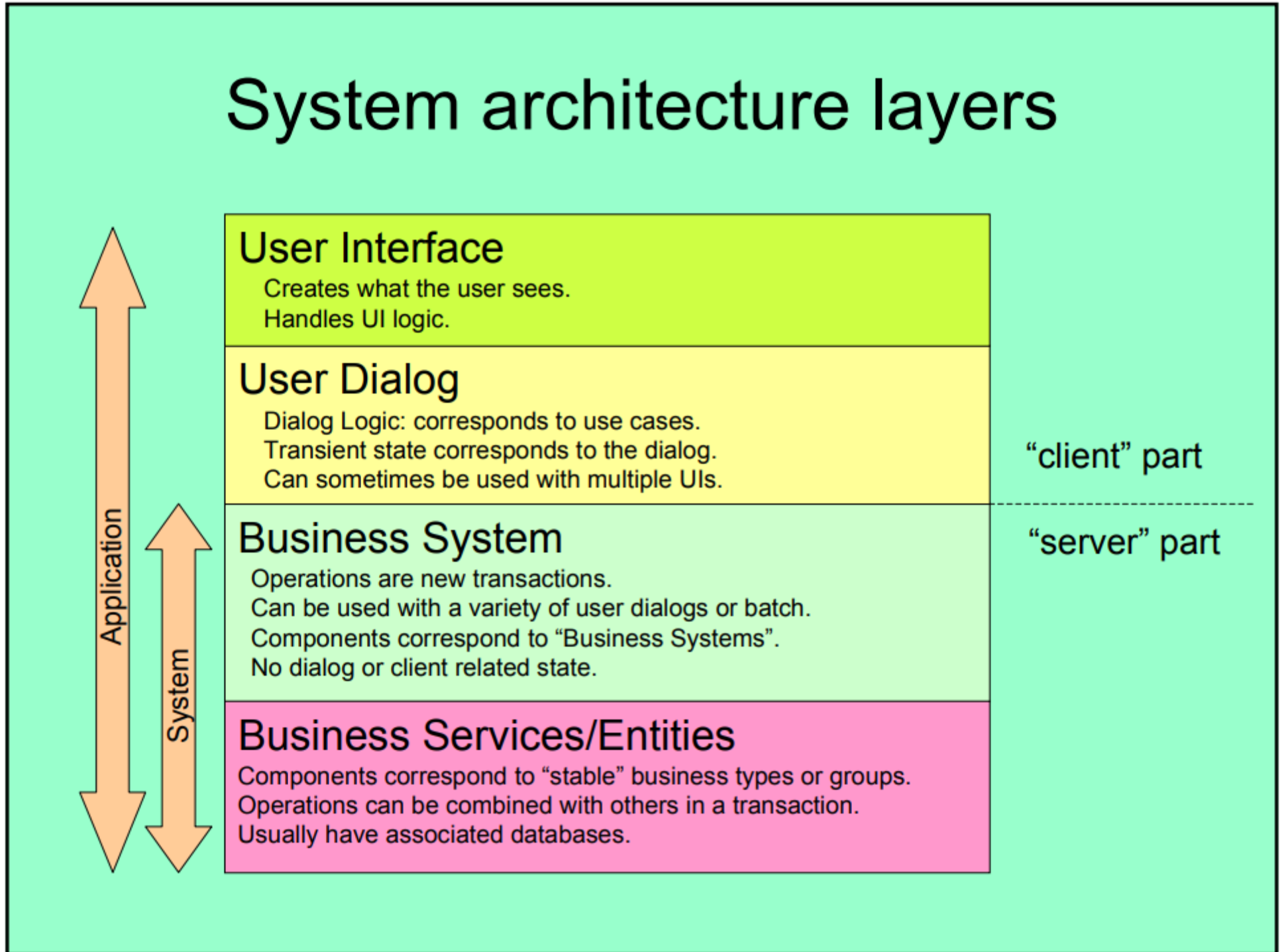
John@Syntropy.co.uk


Requirements
Specification
Component Identification
Component Interaction
Component Specification
Provisioning

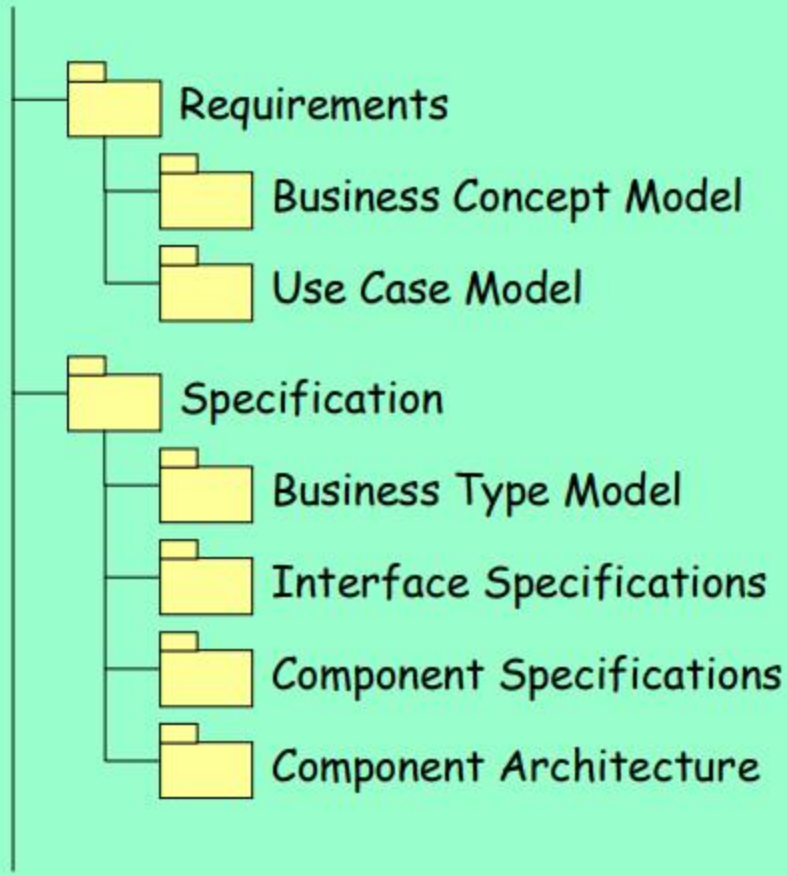
Blueprint for the systems being considered in this tutorial



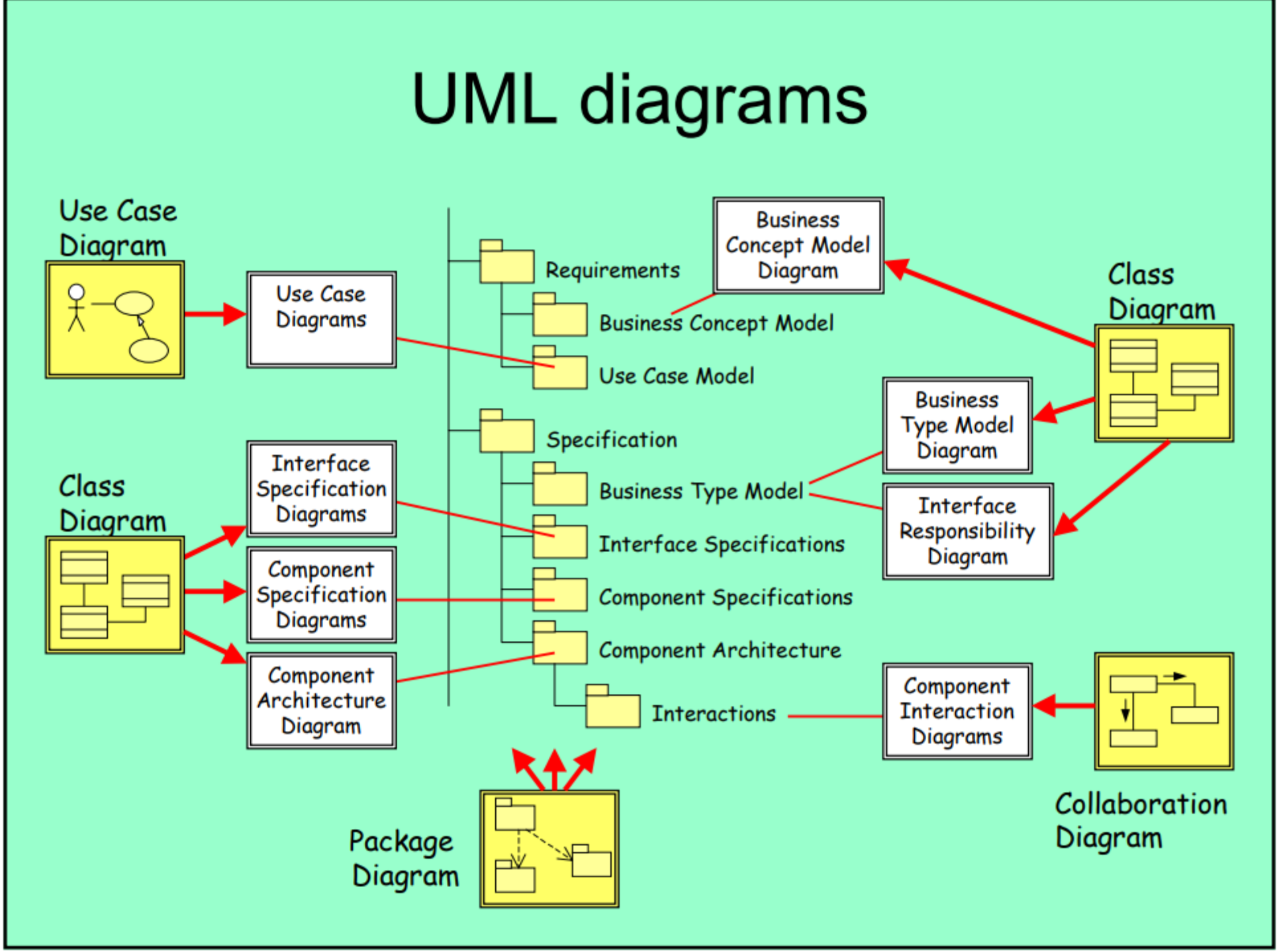
System architecture layers



Organising the artefacts in tool packages

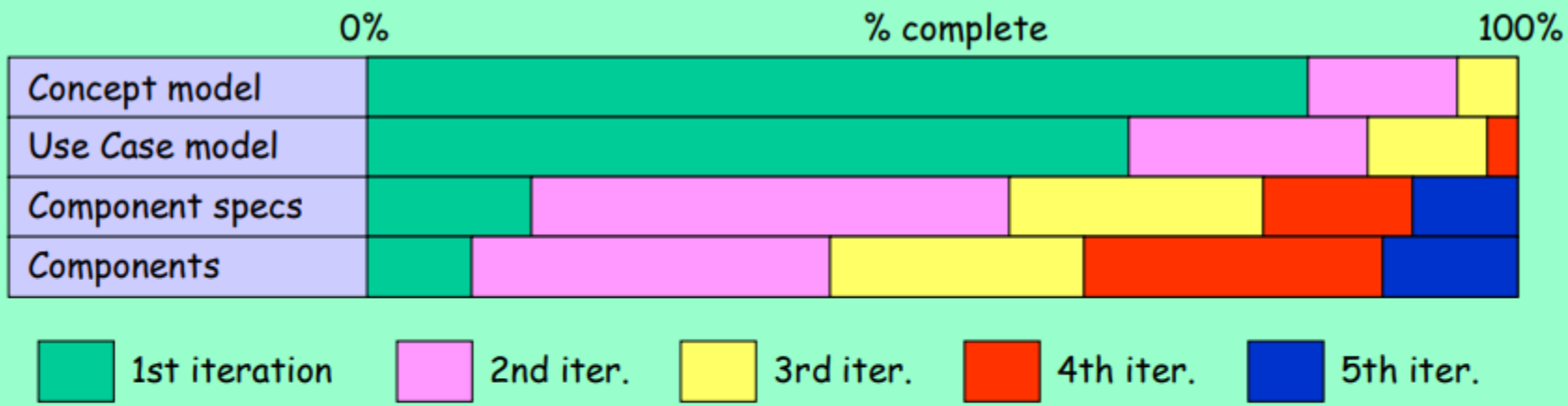


UML diagrams



Evolutionary delivery

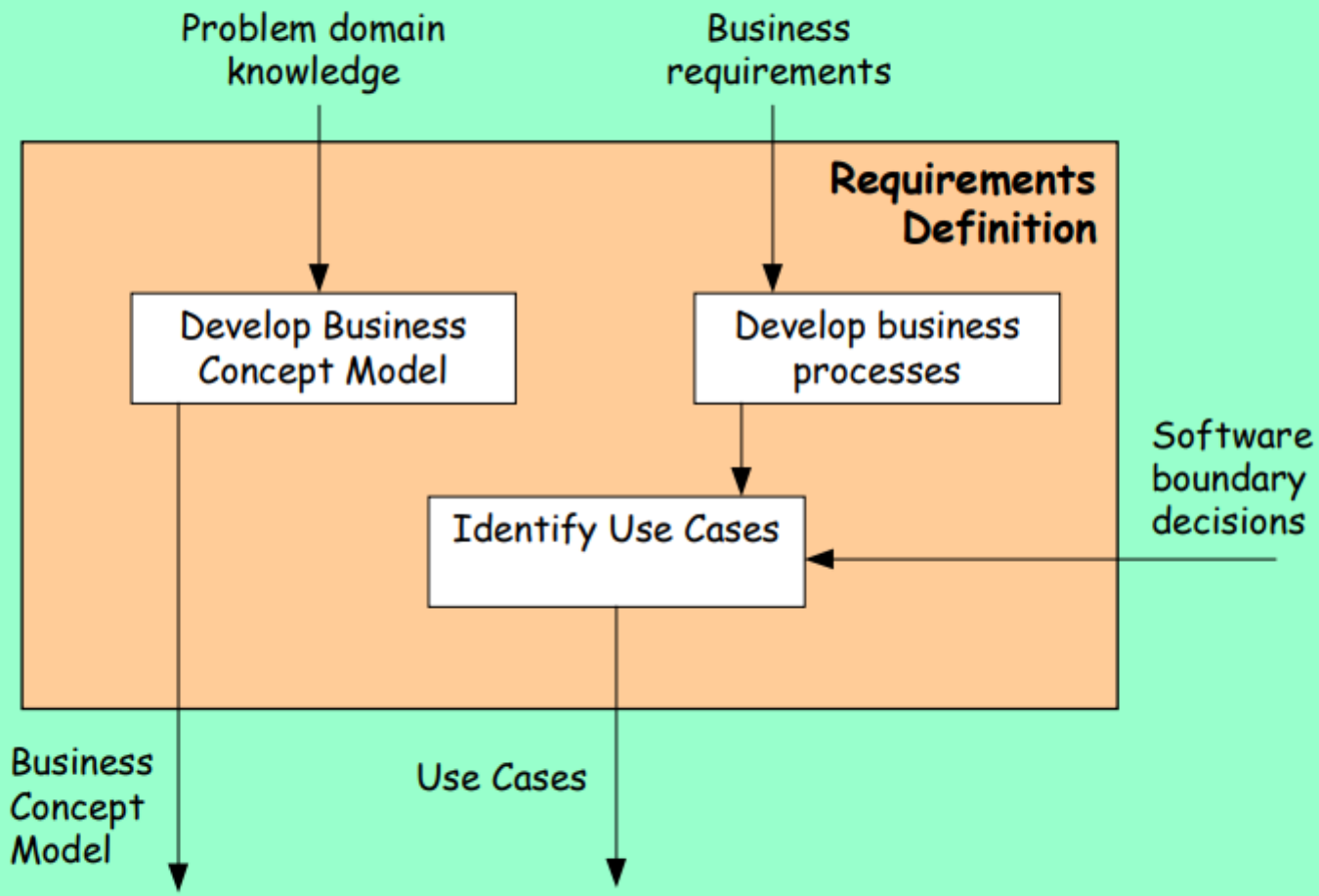
- All the artefacts evolve at each iteration
- Focus is on delivery to the user



Tutorial Map

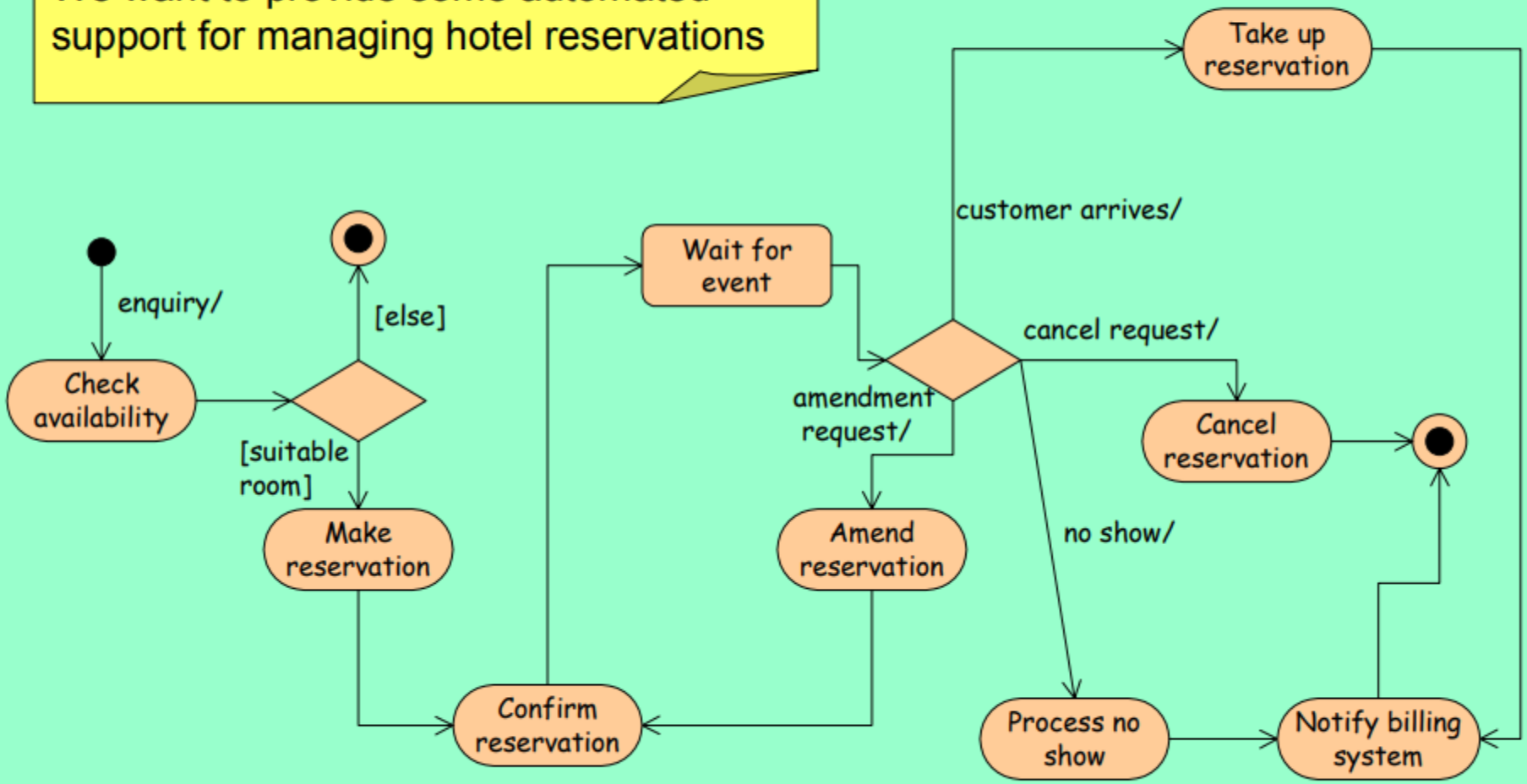
- Requirements Definition
- Component Identification
- Component Interaction
- Component Specification

Requirements Definition

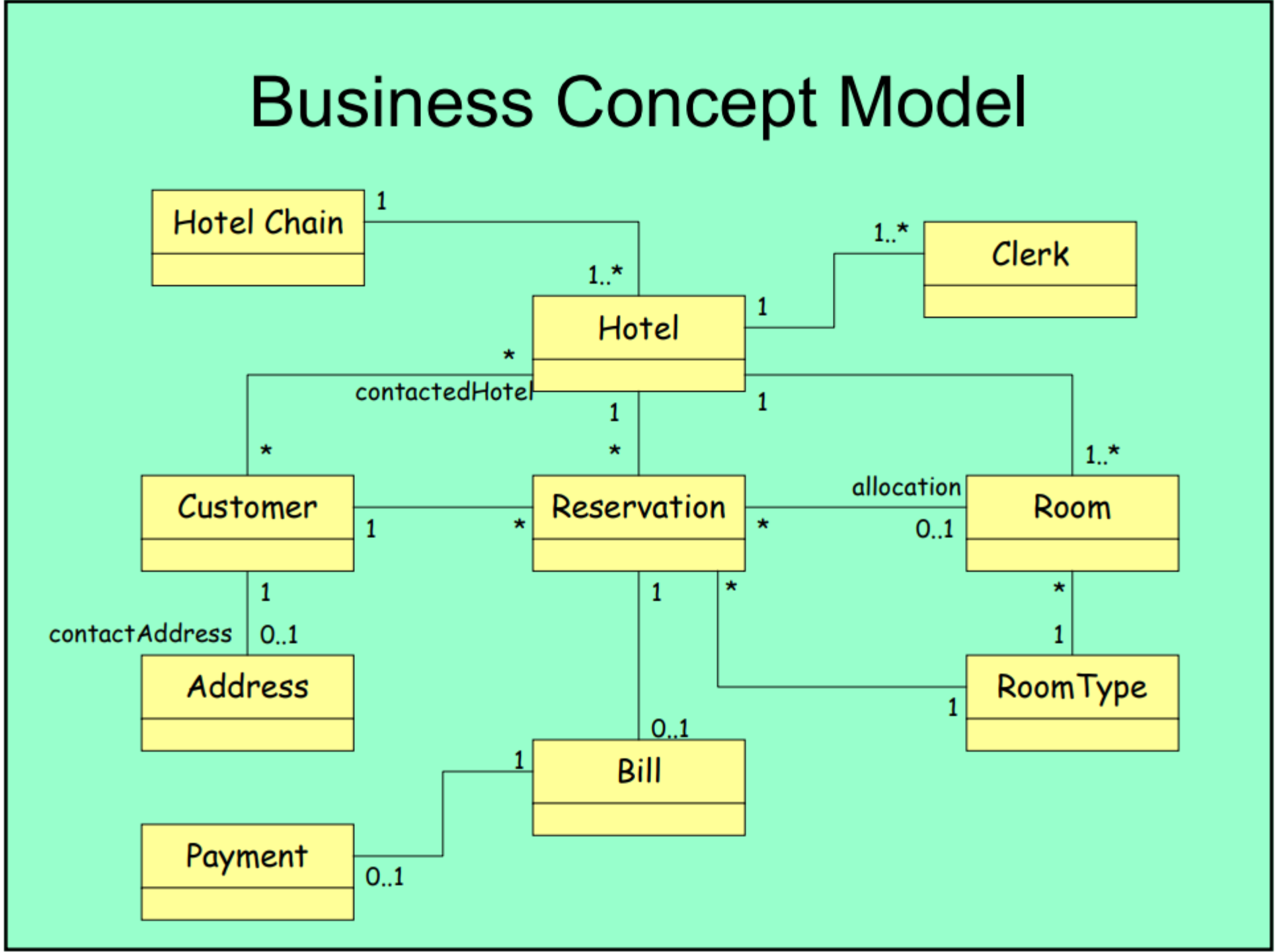


Business process

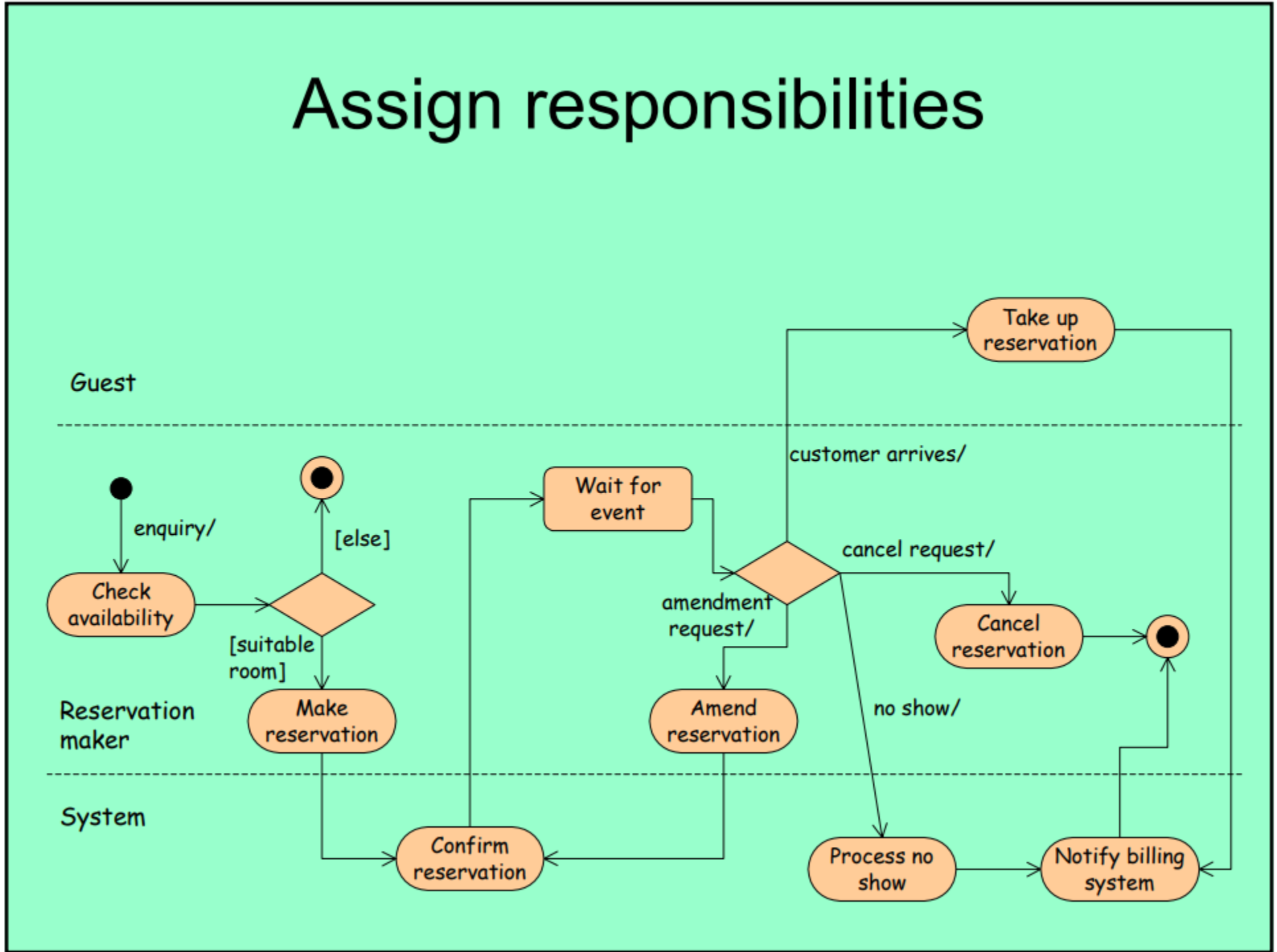
We want to provide some automated support for managing hotel reservations



Business Concept Model

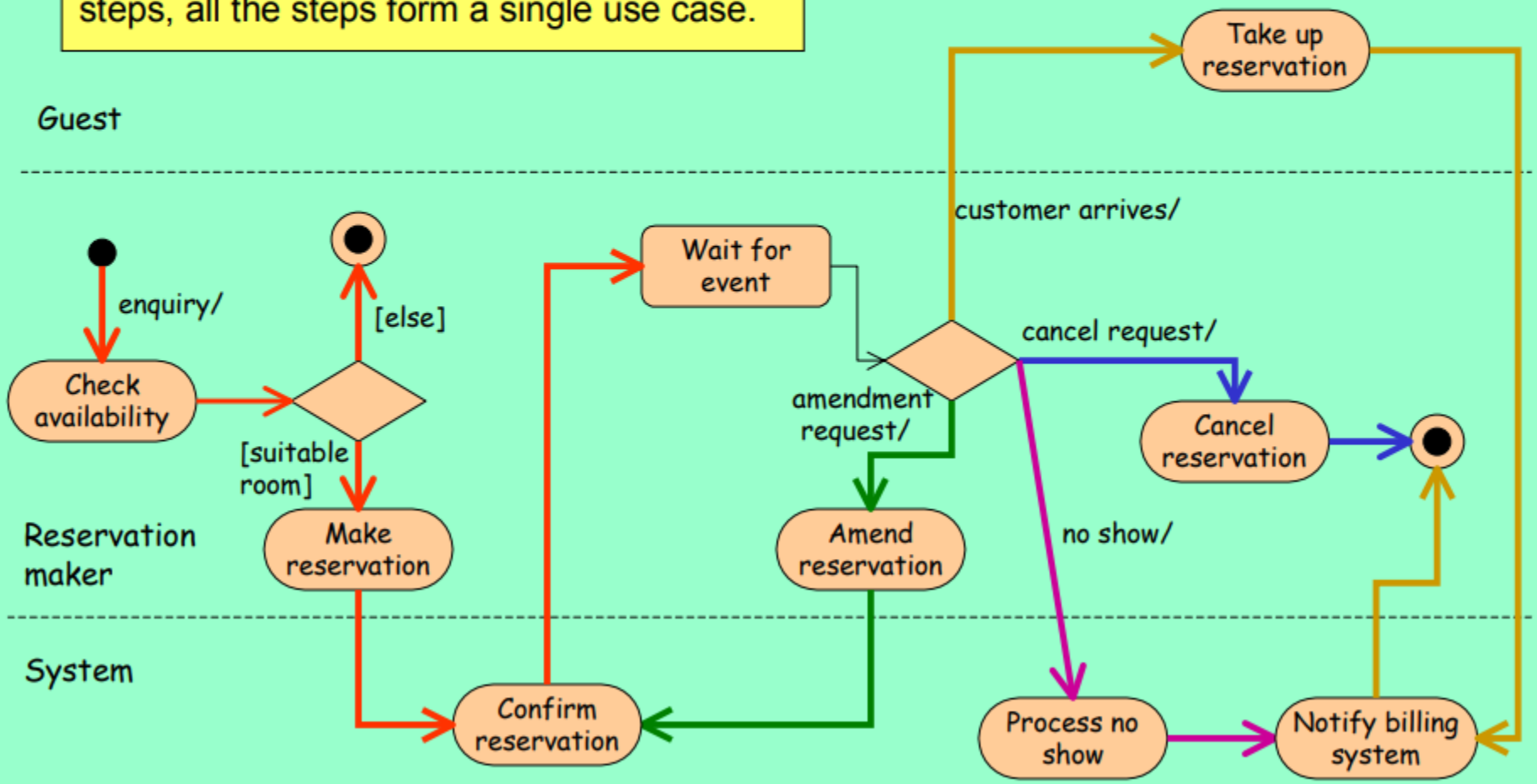


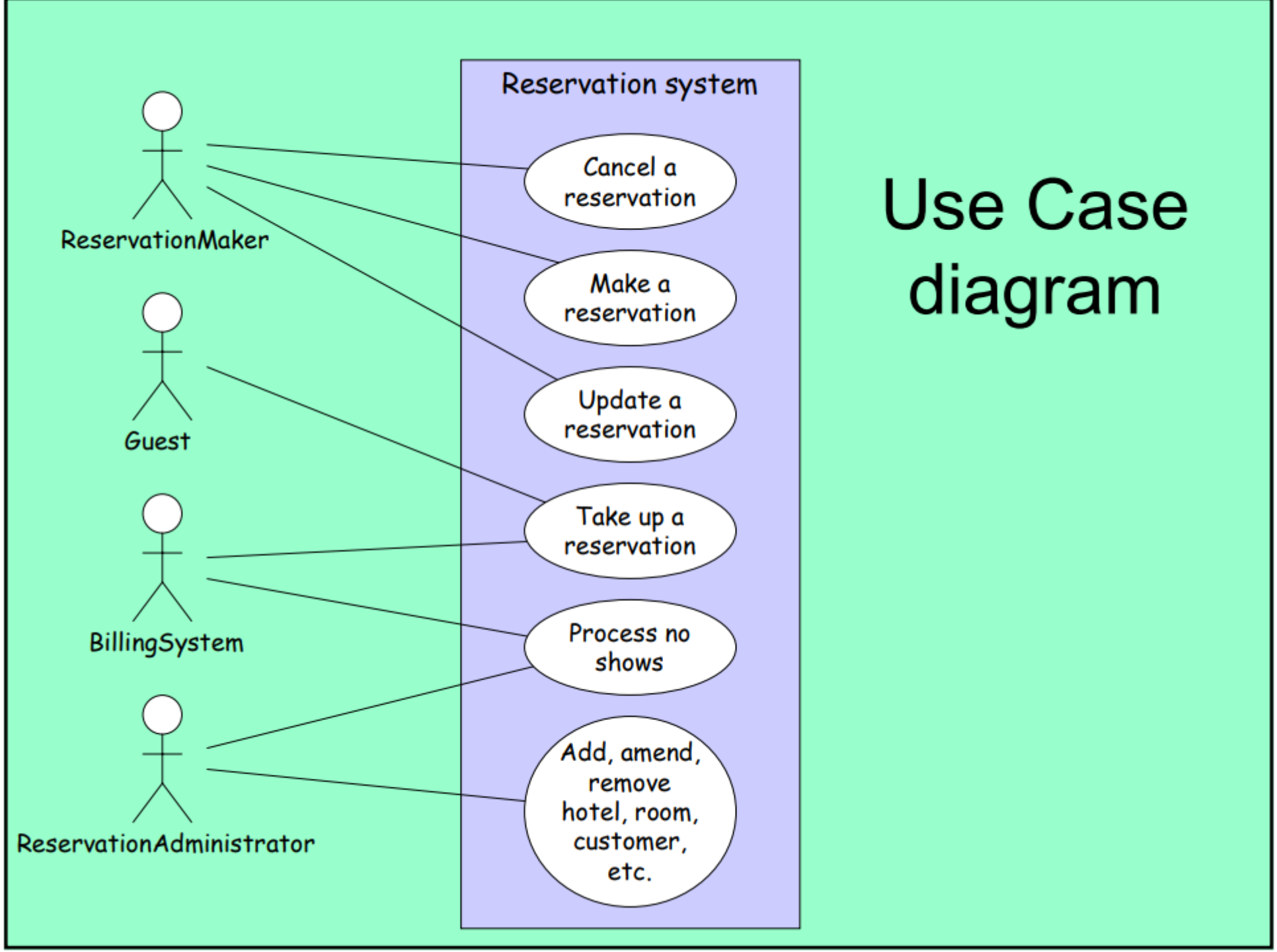
Assign responsibilities



Identify Use Cases

A use case describes the interaction that follows from a single business event. Where an event triggers a number of process steps, all the steps form a single use case.





Use Case diagram

Name	Make a Reservation
Initiator	Reservation Maker
Goal	Reserve a room at a hotel

Steps

or Extension Points

Main success scenario

1. Reservation Maker asks to make a reservation
2. Reservation Maker selects hotel, dates and room type
3. System provides availability and price
4. Reservation Maker agrees to proceed
5. Reservation Maker provides name and postcode
6. Reservation Maker provides contact email address
7. System makes reservation and gives it a tag
8. System reveals tag to Reservation Maker
9. System creates and sends confirmation by email

Extensions

3. Room Not Available
   a) System offers alternative dates and room types
   b) Reservation Maker selects from alternatives

6. Customer already on file
   a) Resume 7

Alternatives
Use an informal "Alternatives" section if you don't want to specify the detail required for an extension

Exercise 1

- Complete the use case on the next slide

Name	Take up a Reservation
Initiator	Guest
Goal	Claim a reservation

Main success scenario

1. Guest arrives at hotel to claim a room
2. Guest provides reservation tag to system
3.

Extensions

Name	Take up a Reservation
Initiator	Guest
Goal	Claim a reservation

Main success scenario

1. Guest arrives at hotel to claim a room
2. Guest provides reservation tag to system
3. System displays reservation details
4. Guest confirms details
5. System allocates a room
6. System notifies billing system that a stay is starting

Extensions

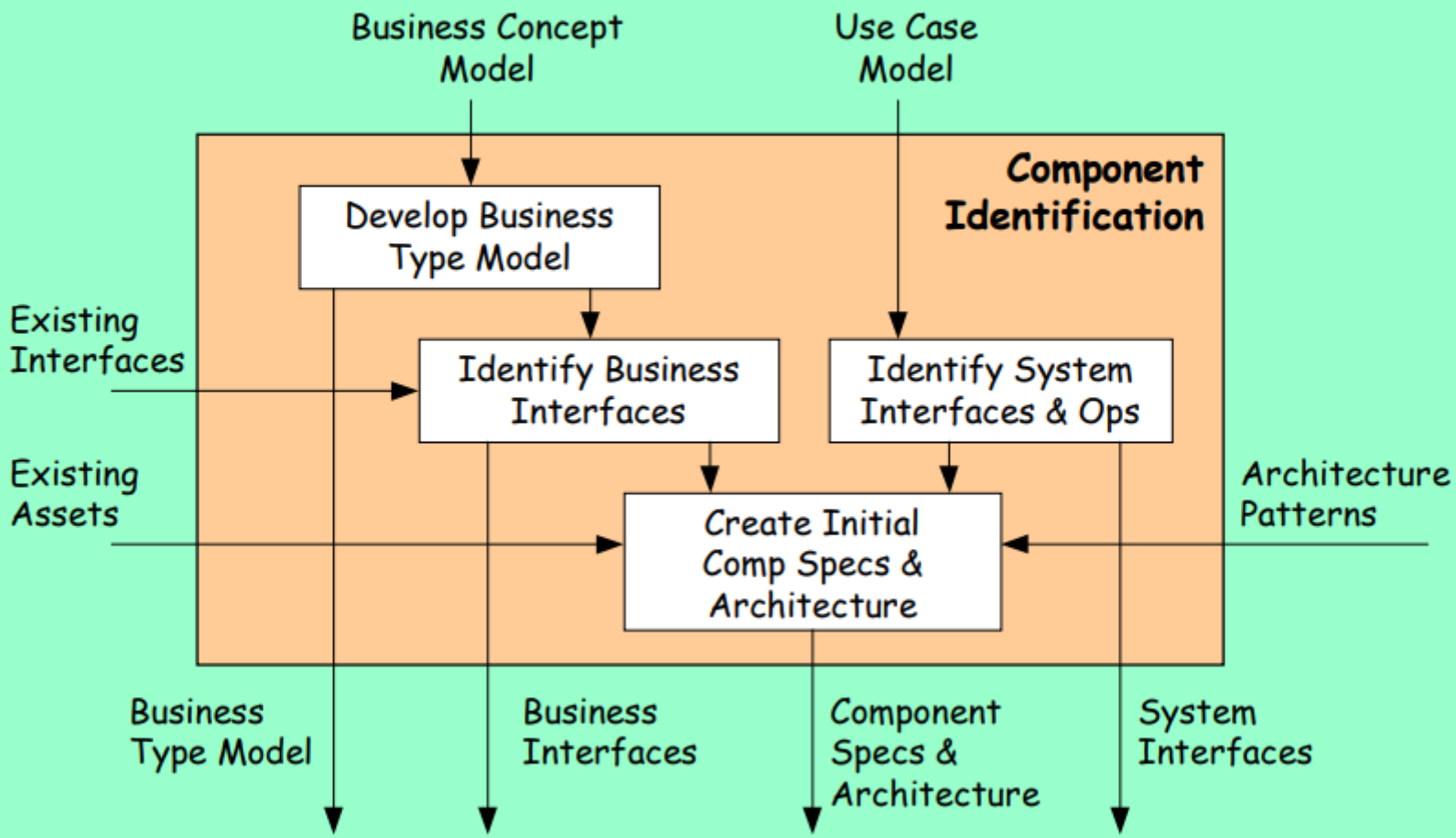
3. Tag not recognised
    1. Fail

etc.

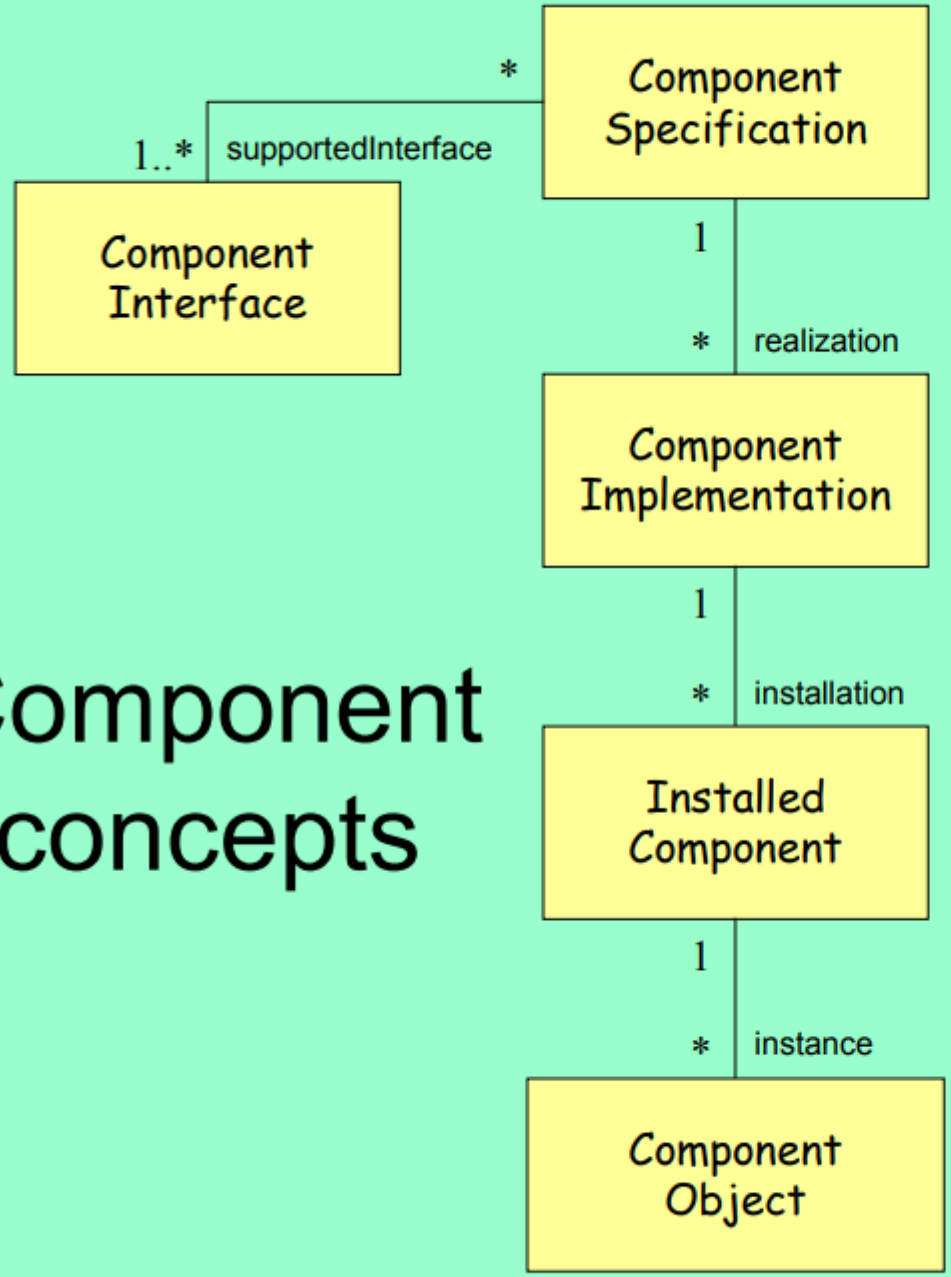
Tutorial Map

- Requirements Definition
- Component Identification
- Component Interaction
- Component Specification

Component Identification

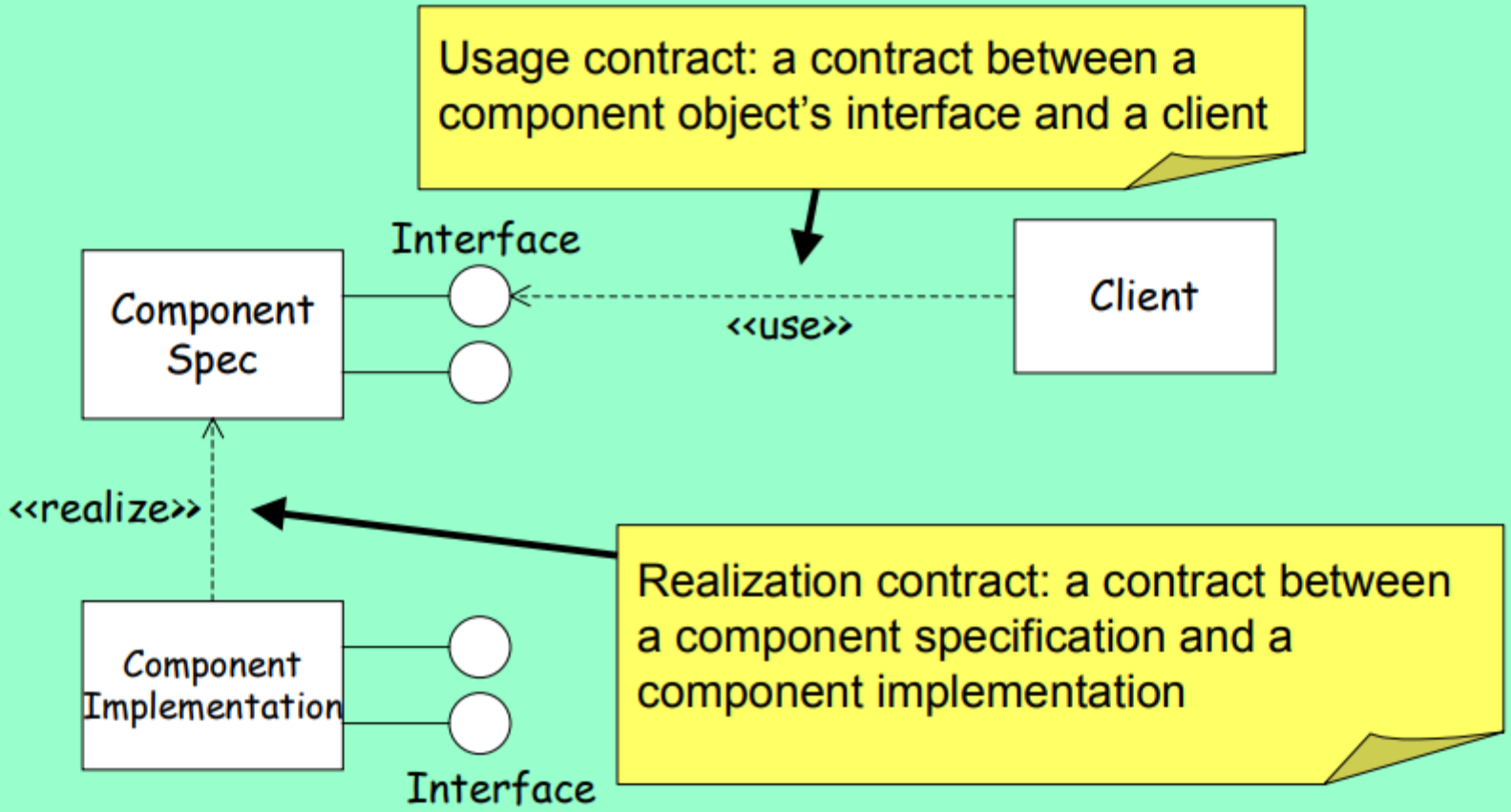


Component concepts

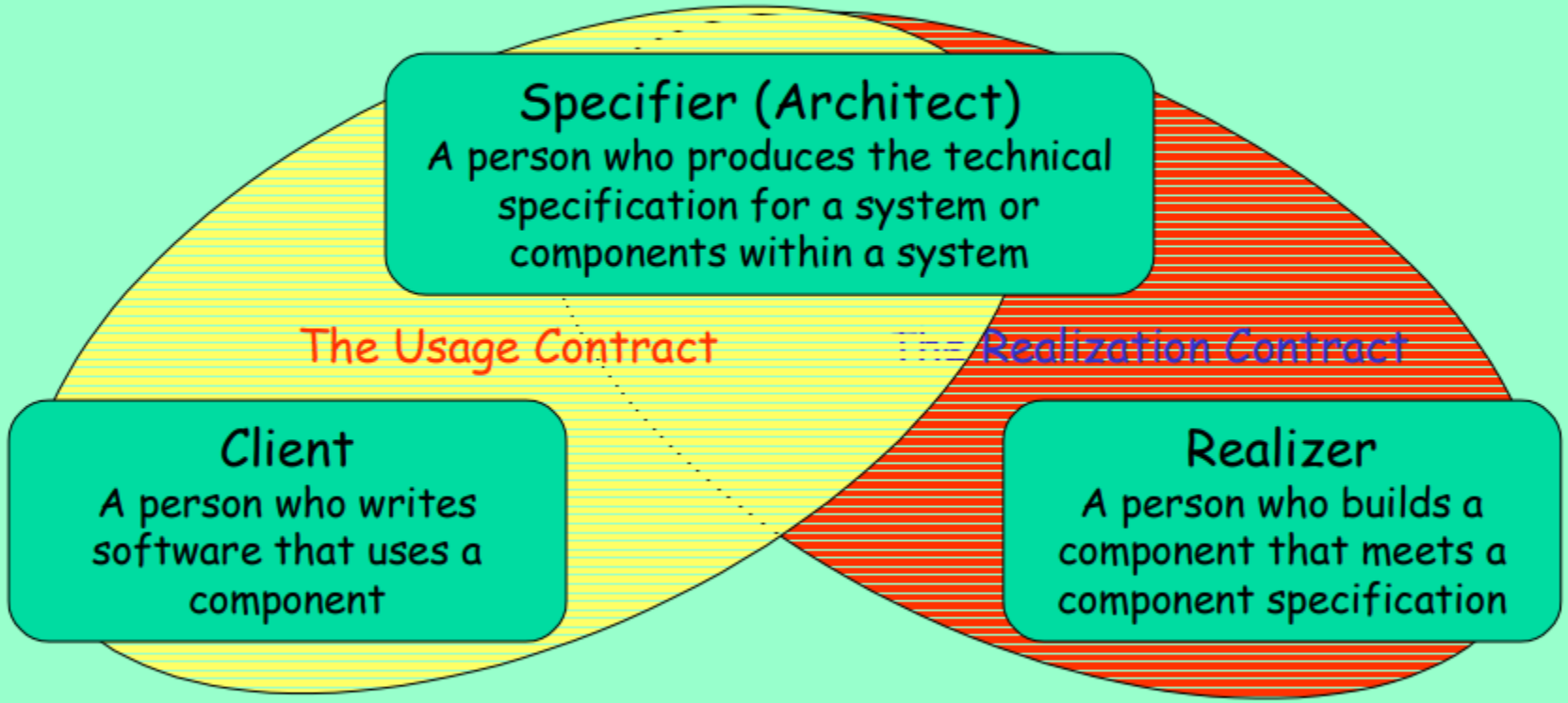


- Component Specification
  - The specification of a unit of software that describes the behaviour of a set of objects, and defines a unit of implementation and deployment
- Component Interface
  - A definition of a set of behaviours that can be offered by a Component Object
- Component Implementation
  - A realization of a Component Specification
- Installed Component
  - An installed (or deployed) copy of a Component Implementation
- Component Object
  - An instance of an Installed Component. A run-time concept

Two distinct contracts



Contracts and roles



Interfaces vs Component Specs

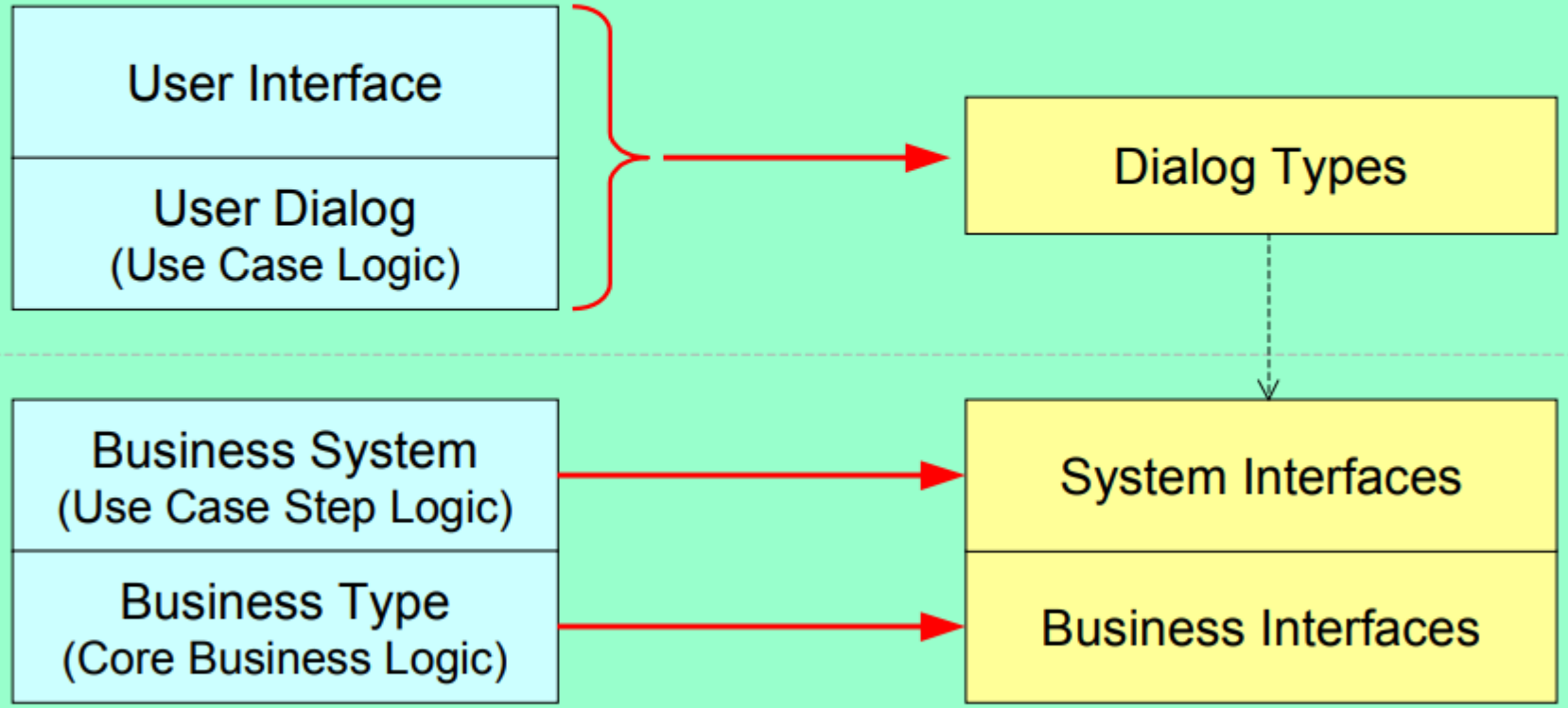
Component Interface

- Represents the usage contract
- Provides a list of operations
- Defines an underlying logical information model specific to the interface
- Specifies how operations affect or rely on the information model
- Describes local effects only

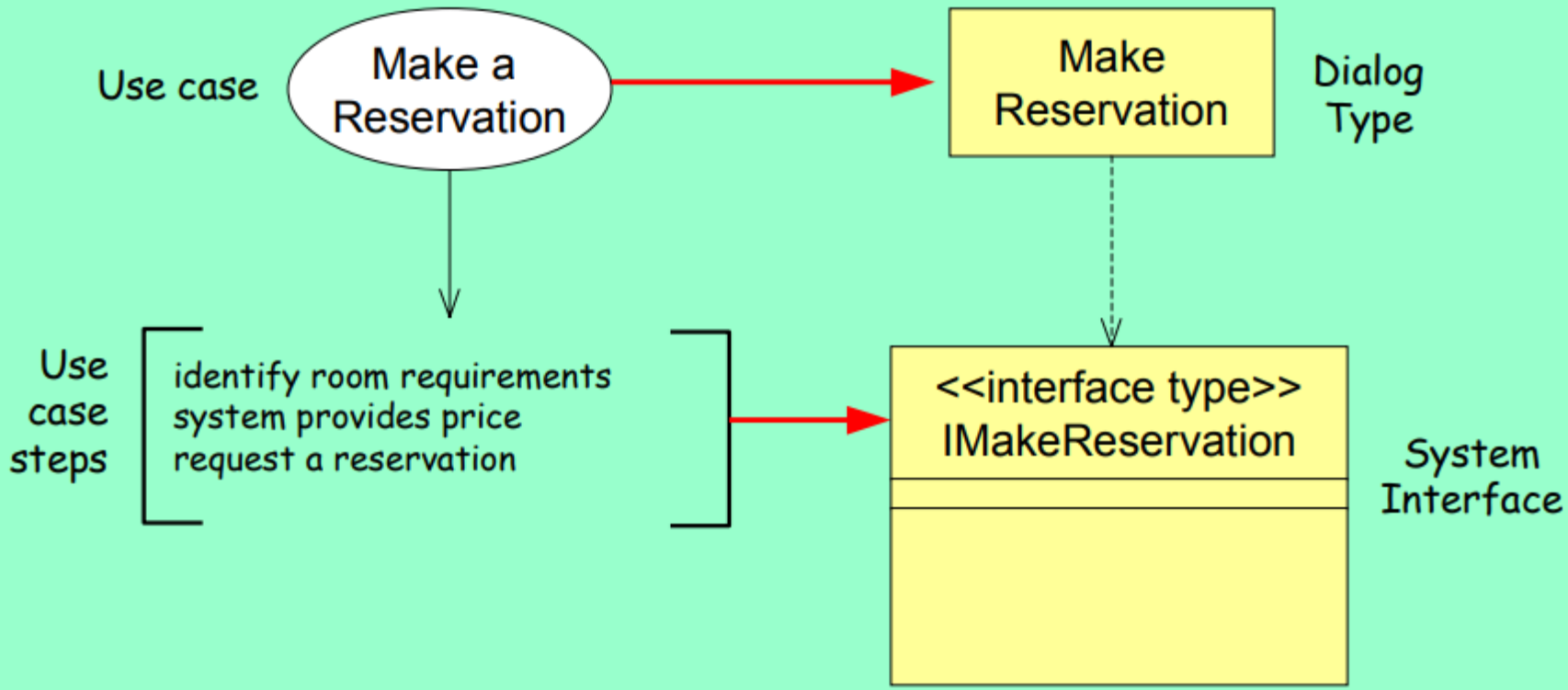
Component Specification

- Represents the realization contract
- Provides a list of supported interfaces
- Defines the run-time unit
- Defines the relationships between the information models of different interfaces
- Specifies how operations should be implemented in terms of usage of other interfaces

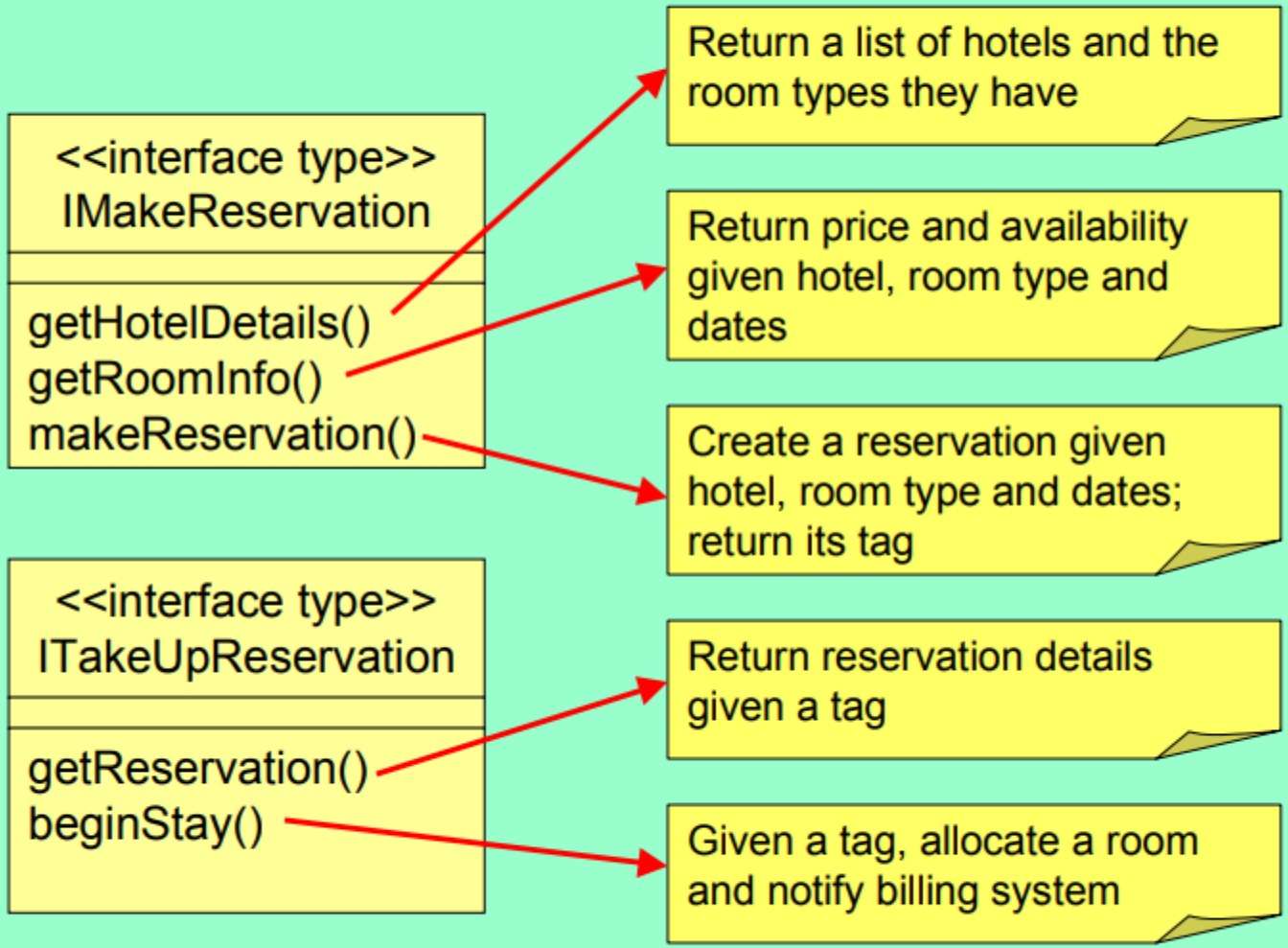
Mapping the architecture layers to software specifications



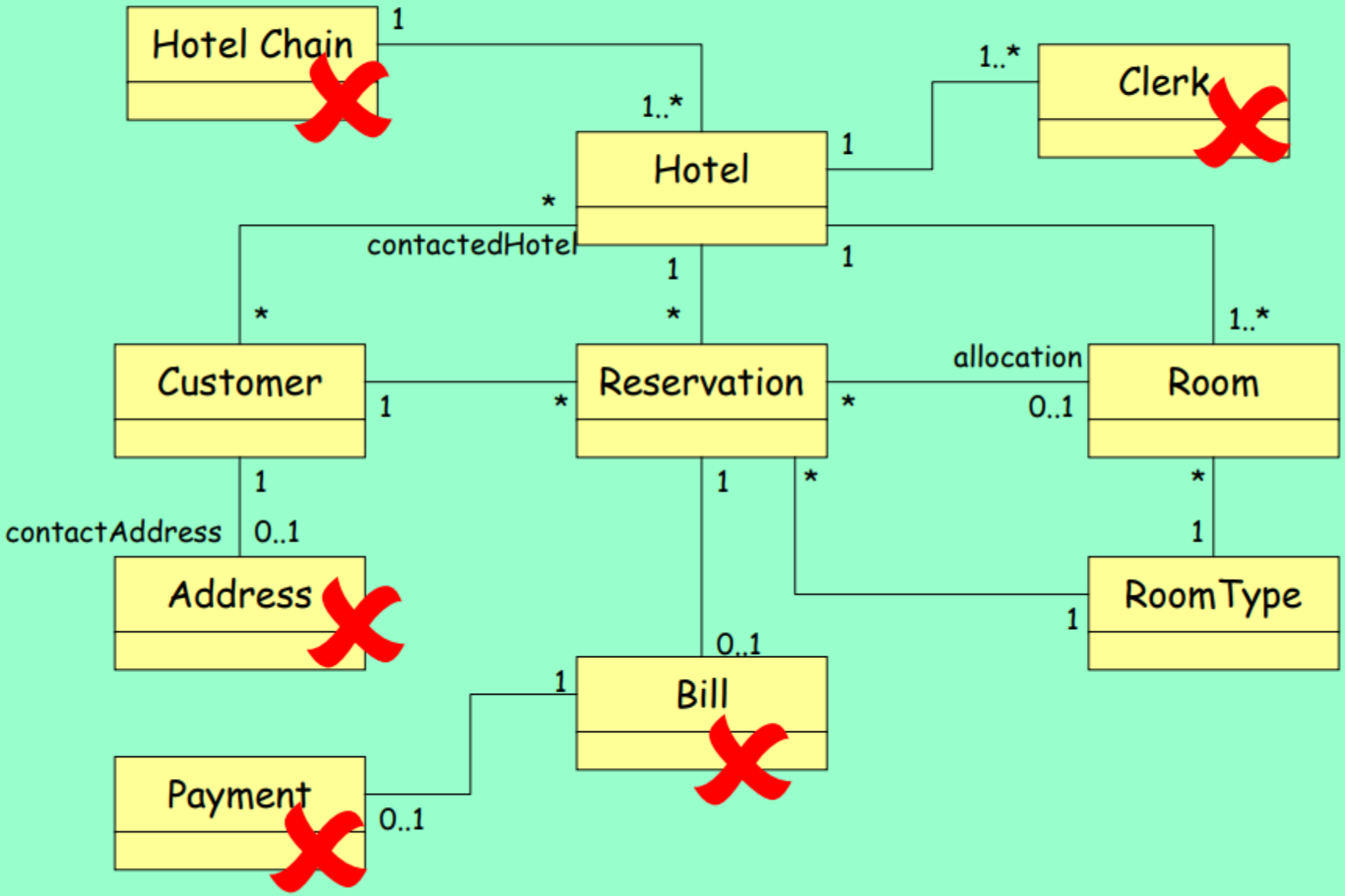
Identify System Interfaces and Operations



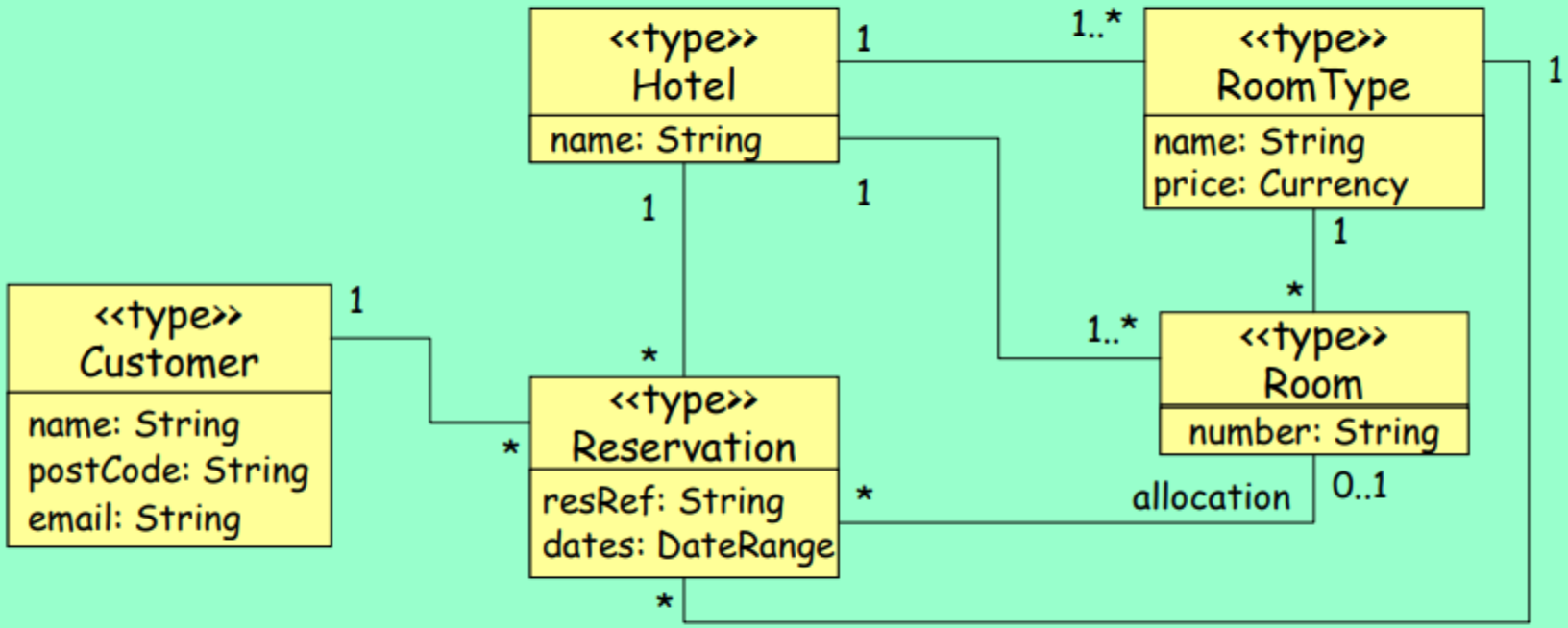
Use case step operations

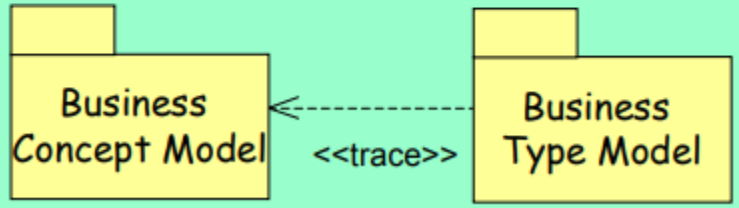


Develop the Business Type Model



Initial Business Type Diagram

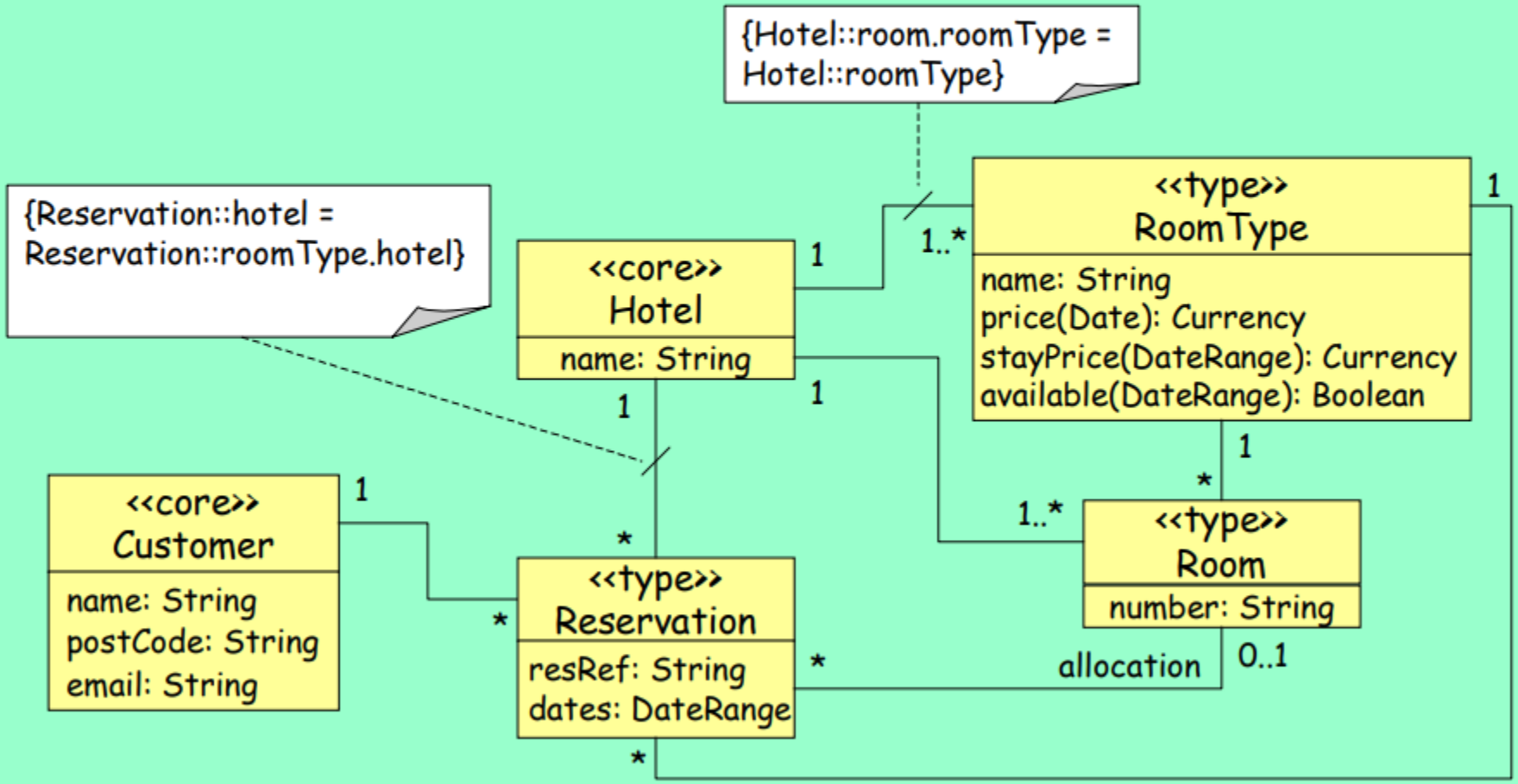




Identify Core types

- Core types represent the primary business information that the system must manage
- Each core type will correspond directly to a business interface
- A core type has:
  - a business identifier, usually independent of other identifiers
  - independent existence – no mandatory associations (multiplicity equal to 1), except to a categorizing type
- In our case study:
  - Customer — YES. Has id (name) and no mandatory assocs.
  - Hotel — YES. Has id (name) and no mandatory assocs.
  - Reservation — NO. Has mandatory assocs.
  - Room — NO. Has mandatory assoc to Hotel
  - RoomType — NO. Has mandatory assoc to Hotel

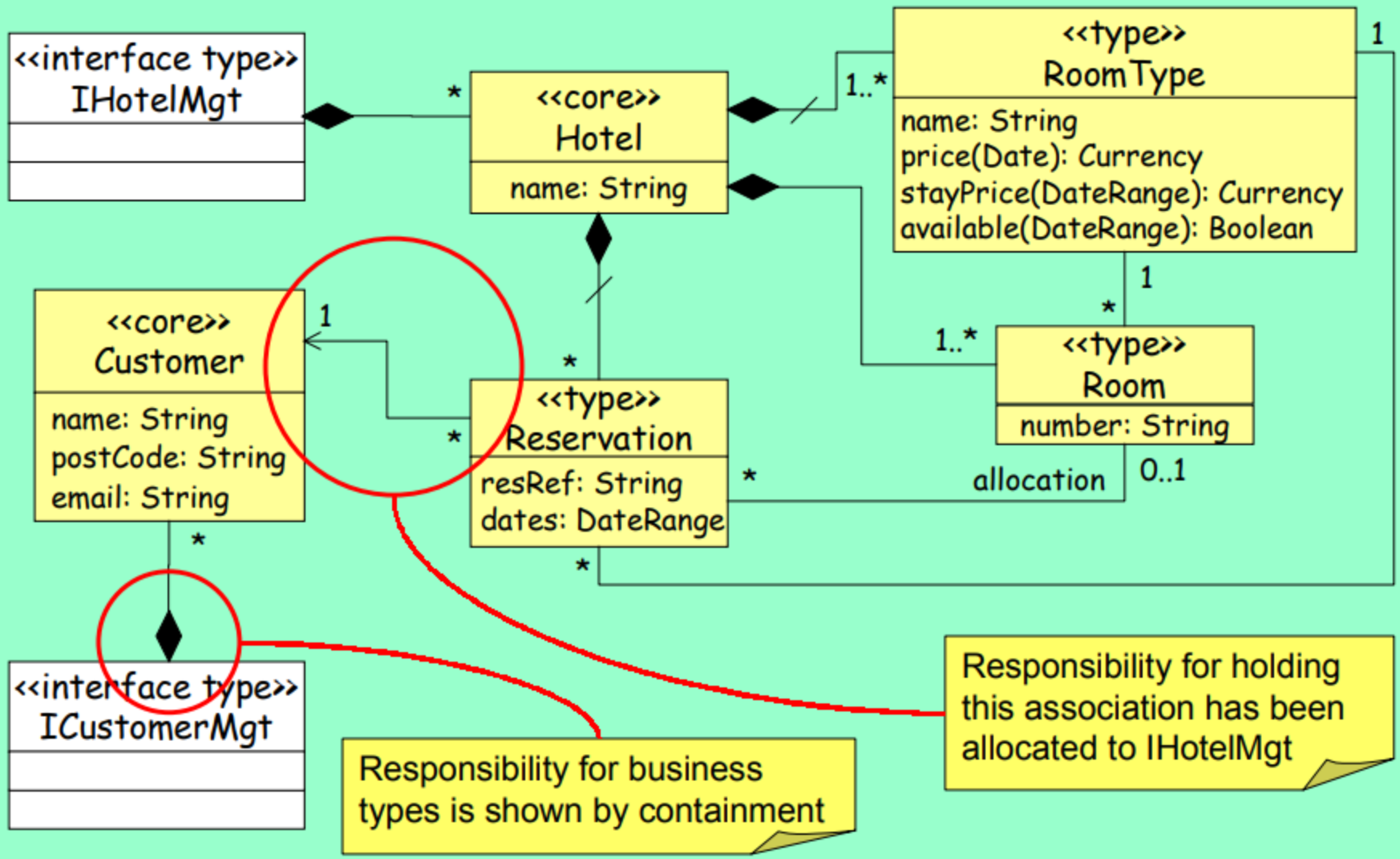
BTM with core types and constraints

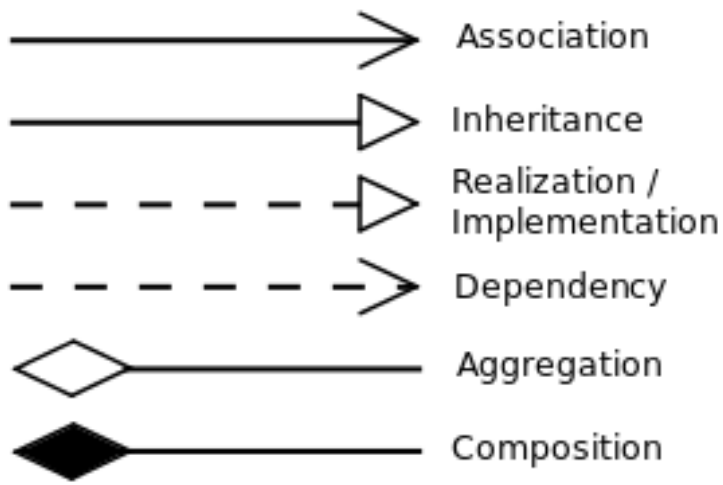


Business rules in the BTM

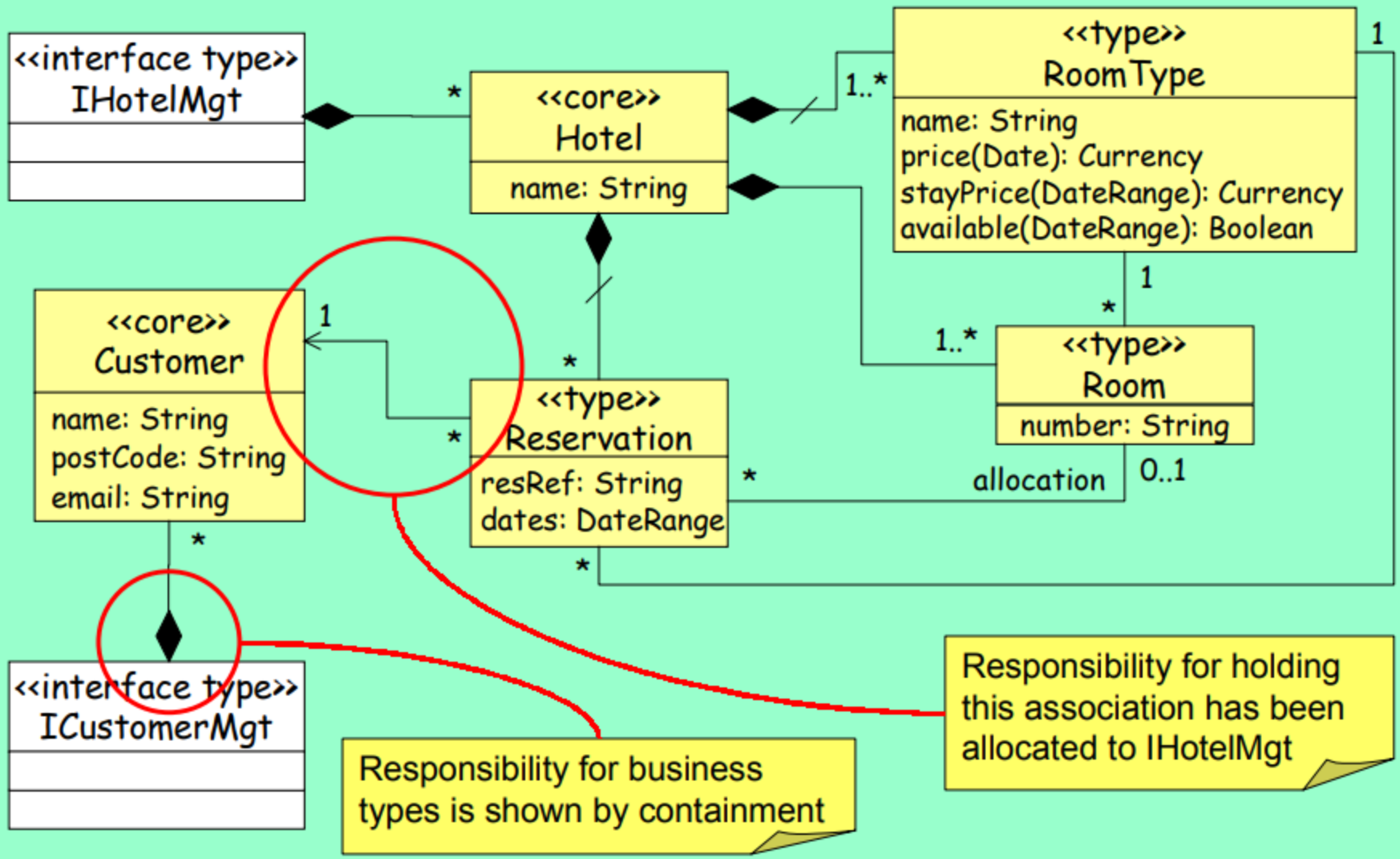
```
context RoomType

-- AVAILABILITY RULES

-- a room is available if the number of rooms reserved on all dates
-- in the range is less than the number of rooms
available(dr) = dr.asSet->collect(d | reservation->select(r | r.allocation->isEmpty and
      r.dates.includes(d))->size)->max < room->size)

-- can never have more reservations for a date than rooms (no overbooking)
Date->forAll(d | reservation->select( r | not r.allocation->isEmpty and
      r.dates.includes(d))->size) <= room->size

-- PRICING RULES

-- the price of a room for a stay is the sum of the prices for the days in the stay
stayPrice(dr) = dr.asSet->collect(d | price(d))->sum
```

Identify business interfaces: The Interface Responsibility Diagram




Association
Inheritance
Realization /
Implementation
Dependency
Aggregation
Composition

Identify business interfaces: The Interface Responsibility Diagram



Component Specifications

- We need to decide what components we want, and which interfaces they will support
- These are fundamental architectural decisions
- Business components:
  - they support the business interfaces
  - remember: components define the unit of development and deployment
- The starting assumption is one component spec per business interface

Component-level Design

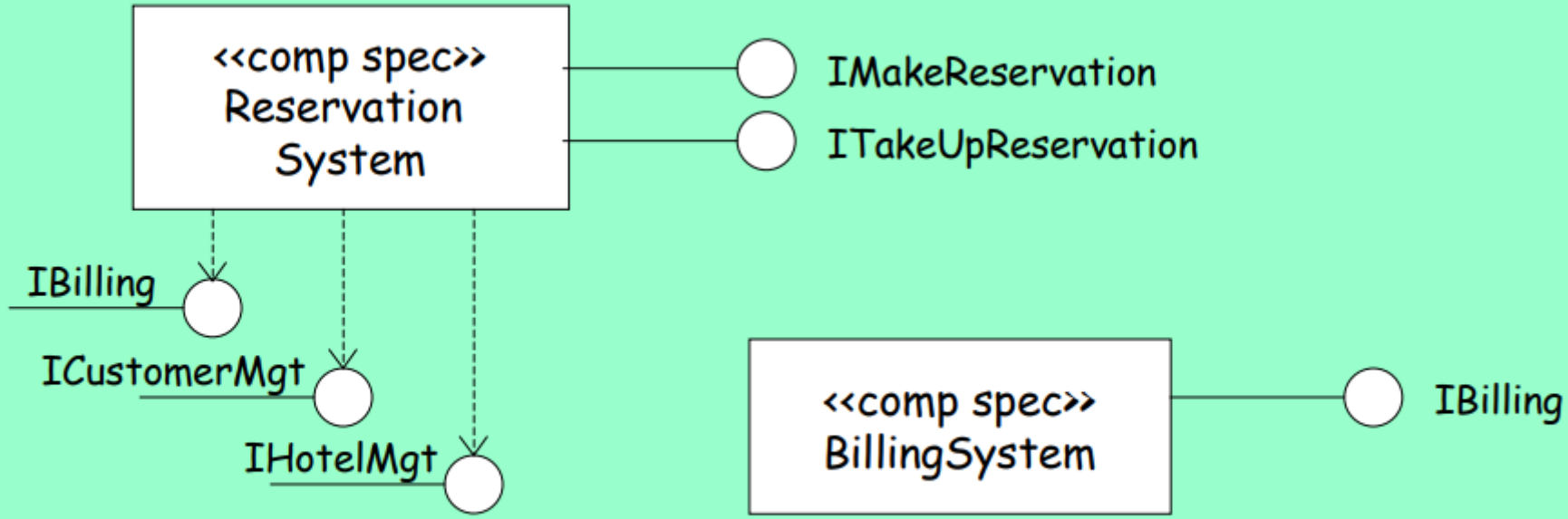
- The Interface Segregation Principle(ISP)

Many client-specific interfaces are better than one general purpose interface.

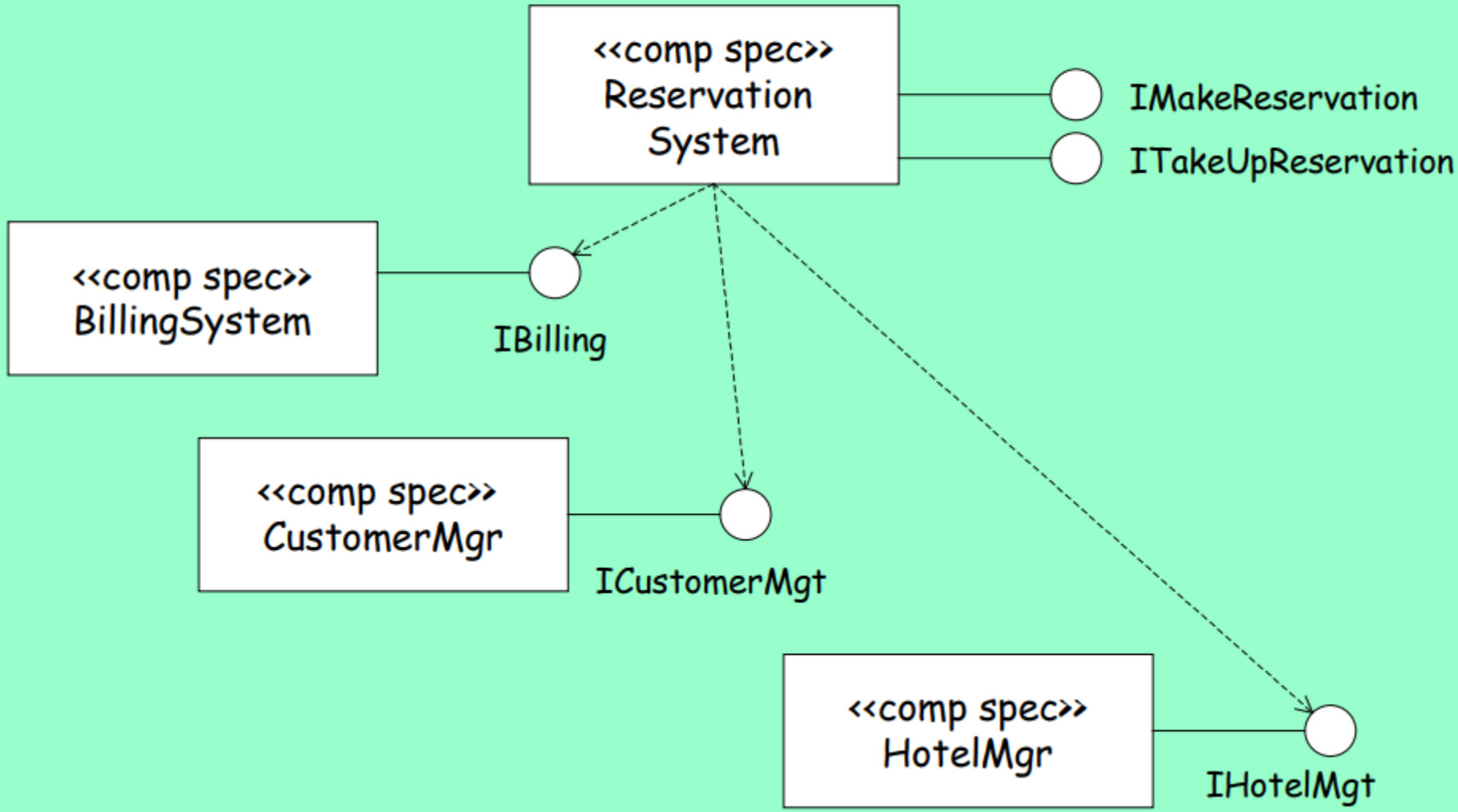
Clients should not be forced to depend upon interfaces that they don't use.

System components

- We will define a single system component spec that supports all the use case system interfaces
  - Alternatives: one component per use case, support system interfaces on the business components
- Separate component spec for billing system wrapper



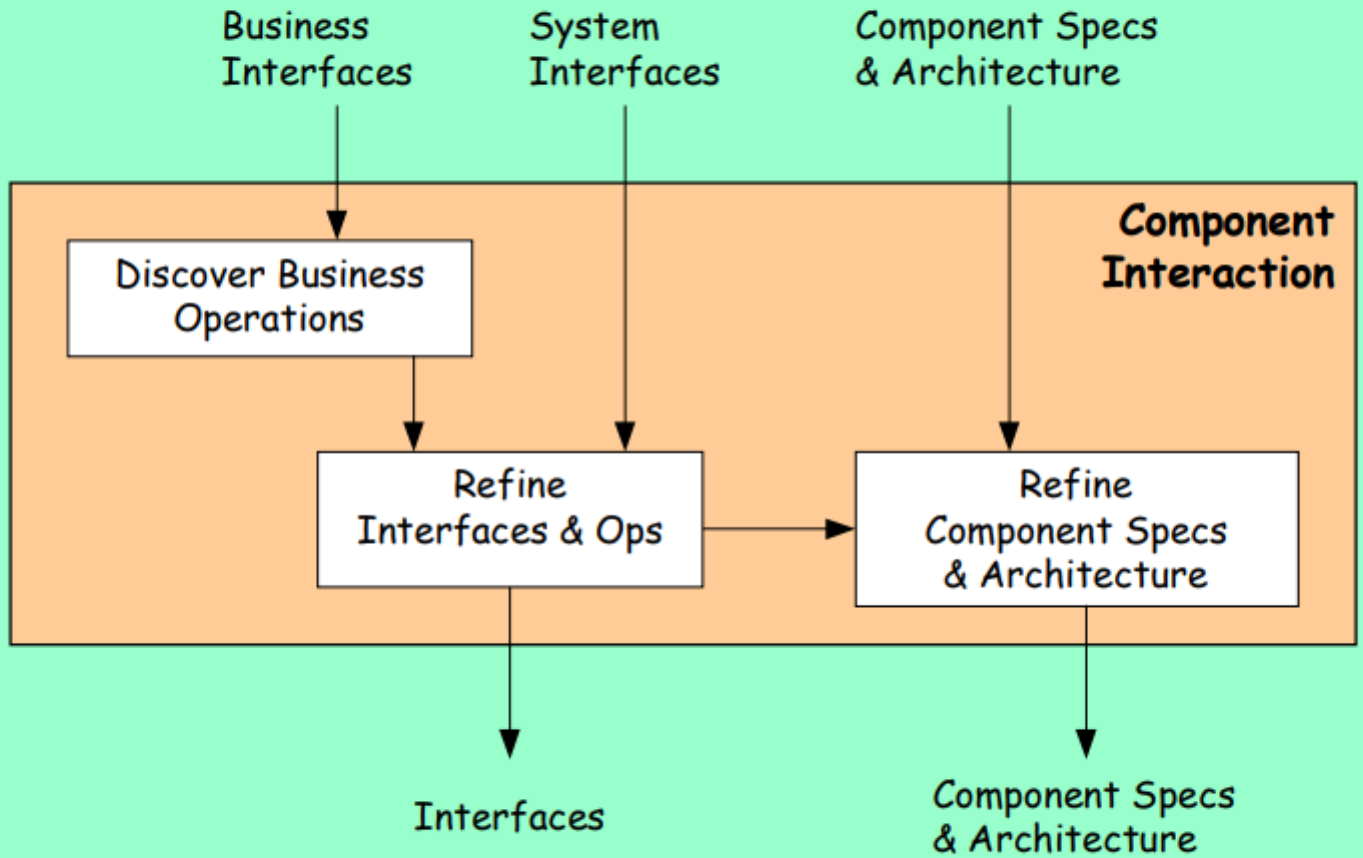
Initial component architecture



Tutorial Map

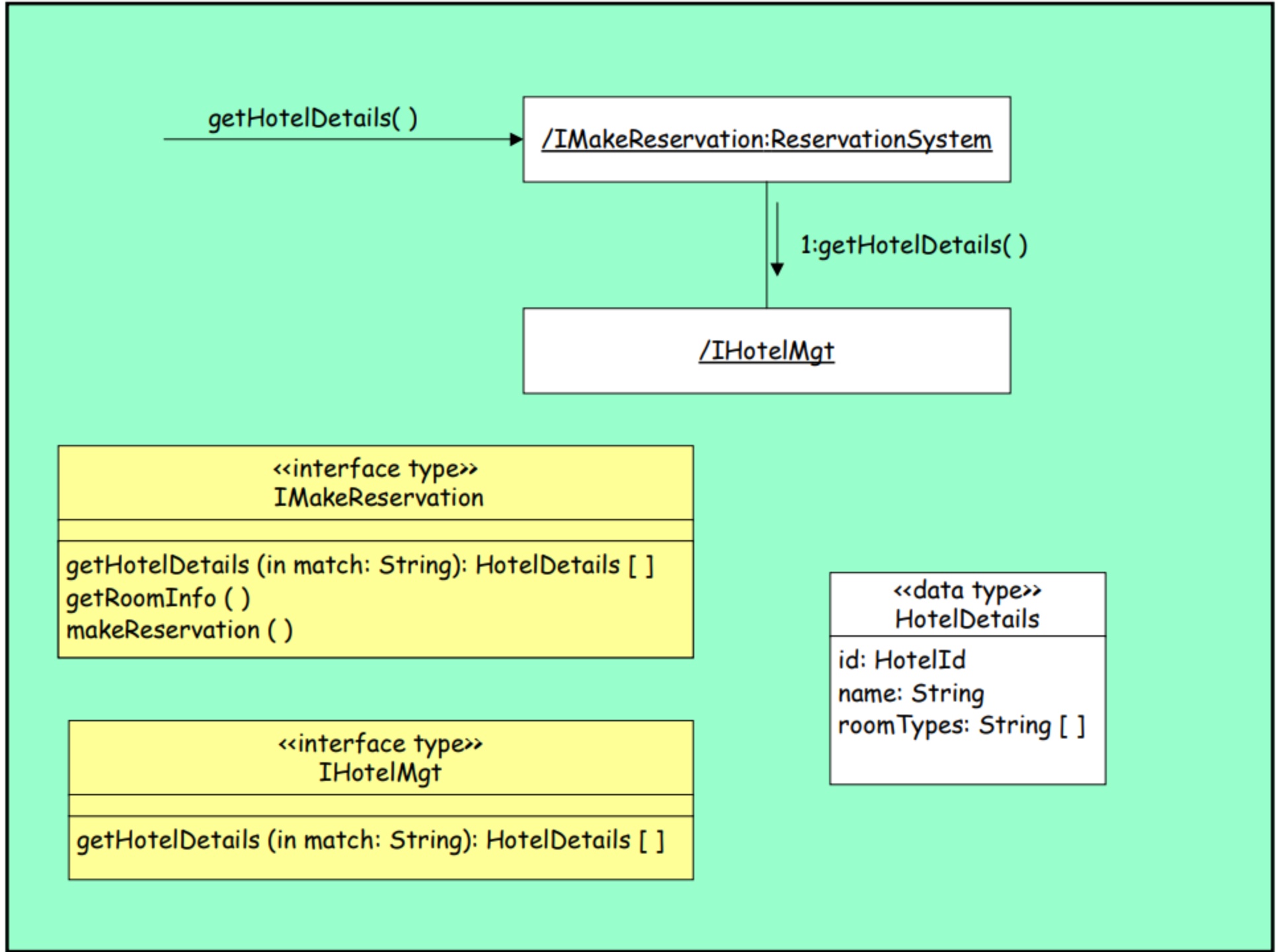
- Requirements Definition
- Component Identification
- Component Interaction
- Component Specification

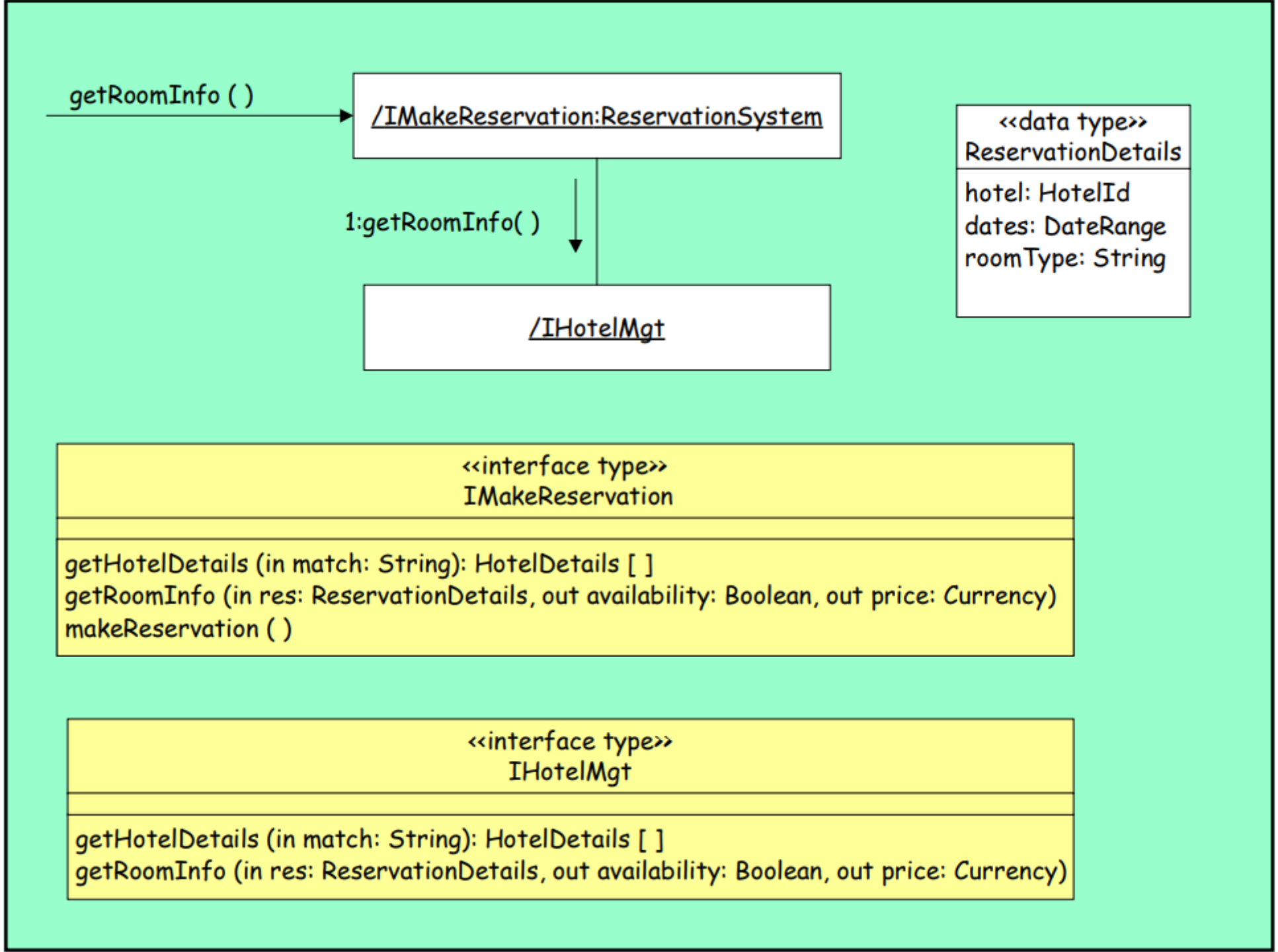
Component Interaction

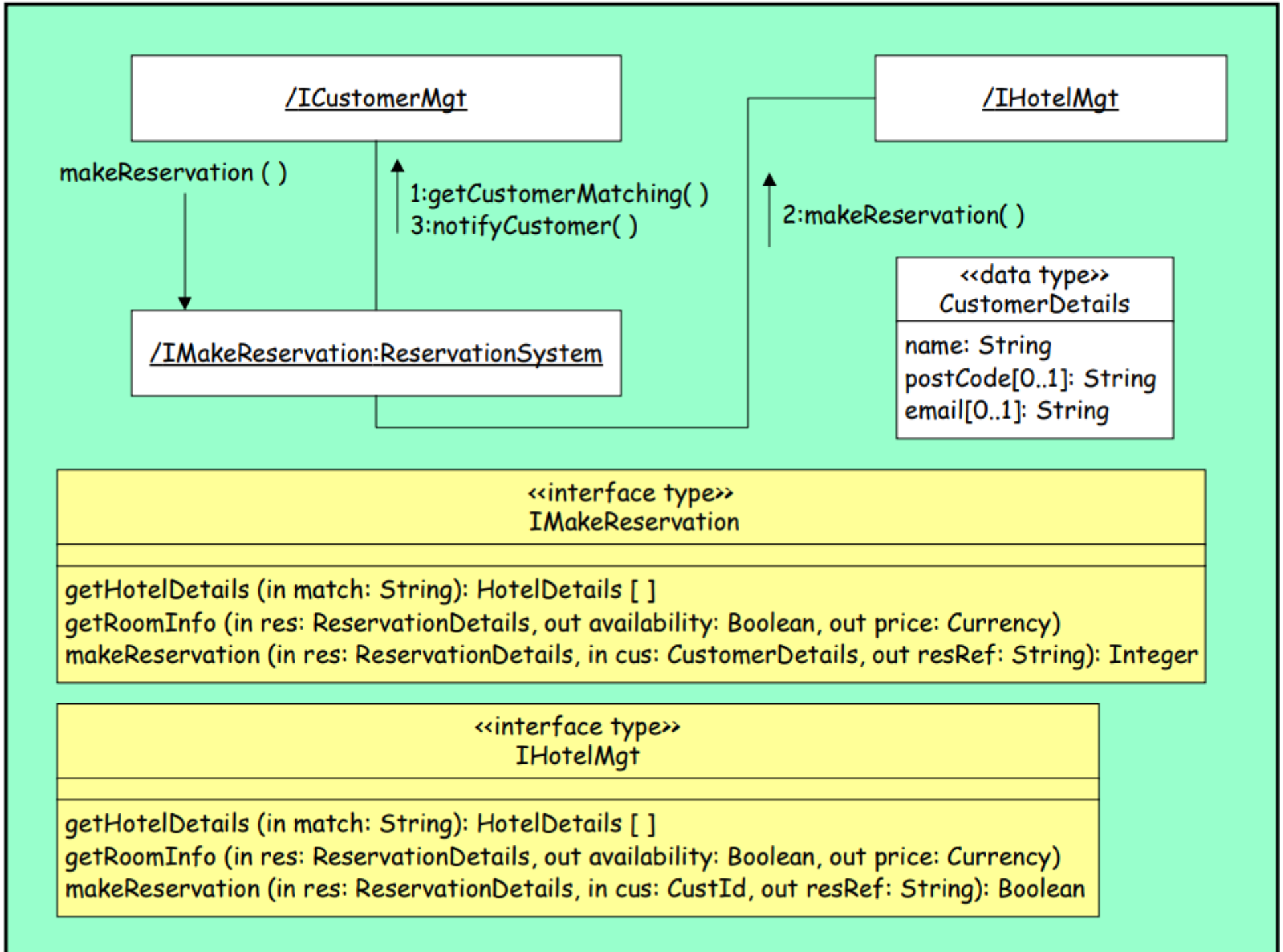


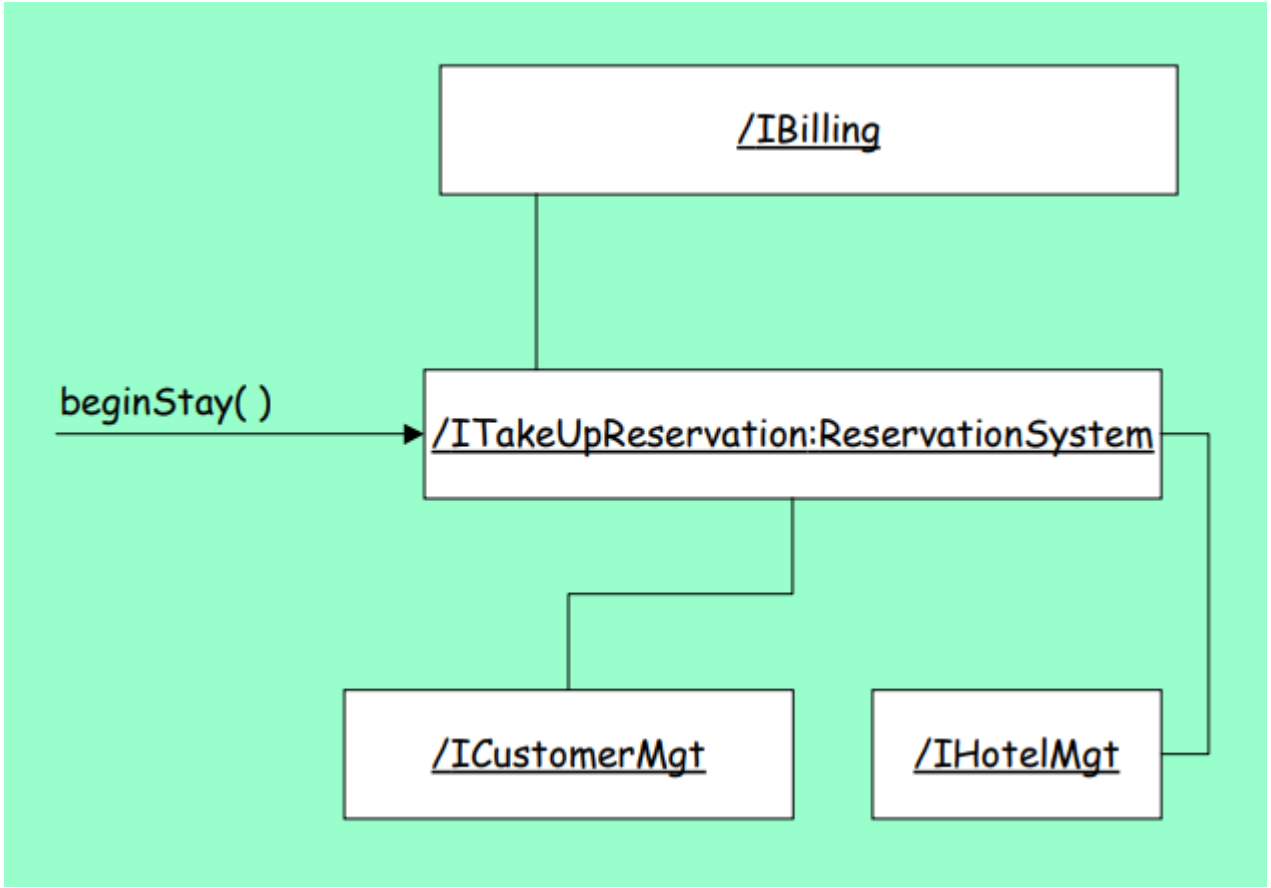
Operation discovery

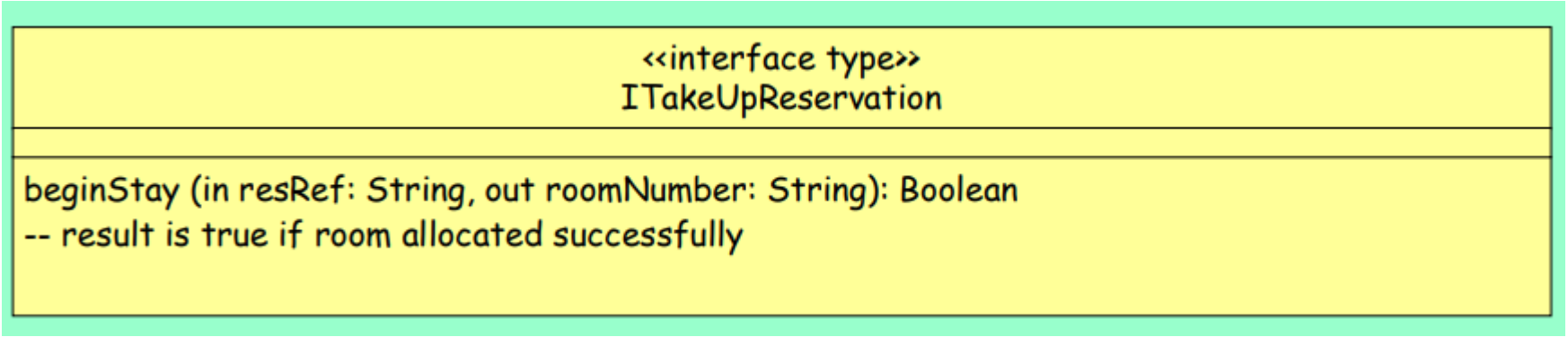
- Uses interaction diagrams (collaboration diagrams)
- The purpose is to discover operations on business interfaces that must be specified
  - not all operations will be discovered or specified
- Take each use case step operation in turn:
  - decide how the component offering it should interact with components offering the business interfaces
  - draw one or more collaboration diagram per operation
  - define signatures for all operations

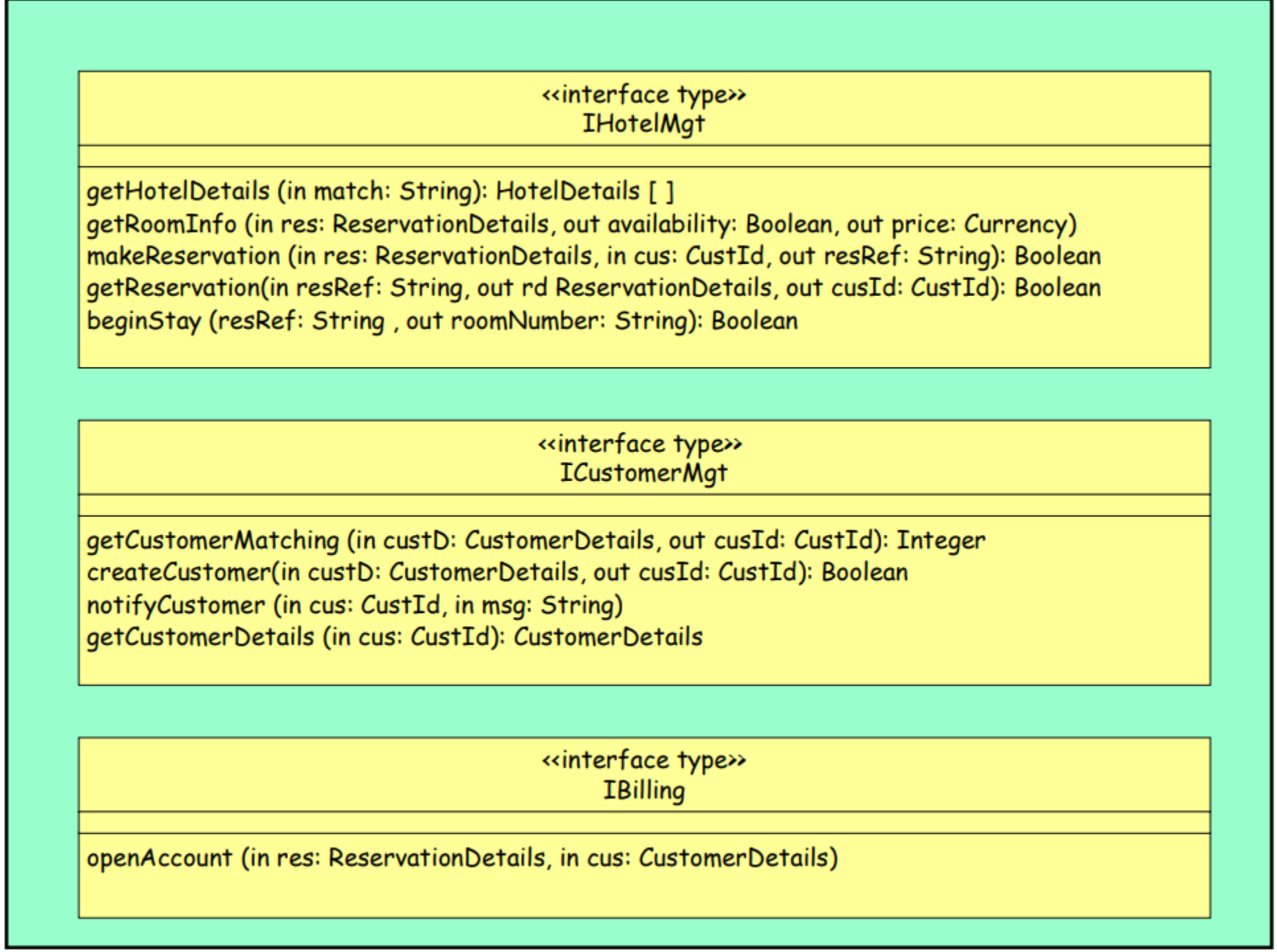

getHotelDetails( )
/IMakeReservation:ReservationSystem
1:getHotelDetails( )
/IHotelMgt
<<interface type>>
IMakeReservation
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo ( )
makeReservation ( )
<<data type>>
HotelDetails
id: HotelId
name: String
roomTypes: String [ ]
<<interface type>>
IHotelMgt
getHotelDetails (in match: String): HotelDetails [ ]

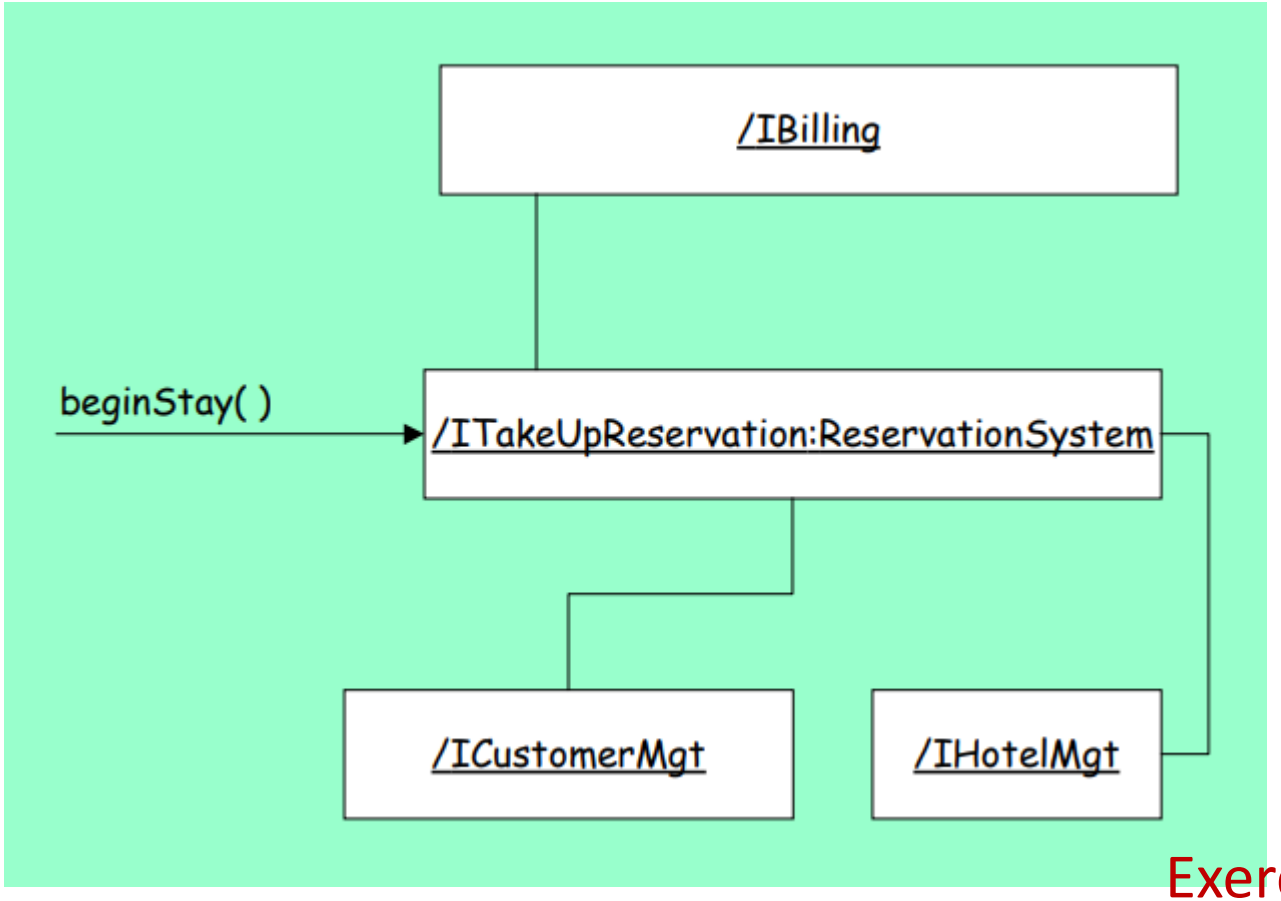

getRoomInfo ( )
/IMakeReservation:ReservationSystem
1:getRoomInfo( )
/IHotelMgt
<<data type>>
ReservationDetails
hotel: HotelId
dates: DateRange
roomType: String
<<interface type>>
IMakeReservation
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation ( )
<<interface type>>
IHotelMgt
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)


/ICustomerMgt
/IHotelMgt
makeReservation ( )
1:getCustomerMatching( )
3:notifyCustomer( )
2:makeReservation( )
/IMakeReservation:ReservationSystem
<<data type>>
CustomerDetails
name: String
postCode[0..1]: String
email[0..1]: String
<<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>>
IHotelMgt
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation (in res: ReservationDetails, in cus: CustId, out resRef: String): Boolean

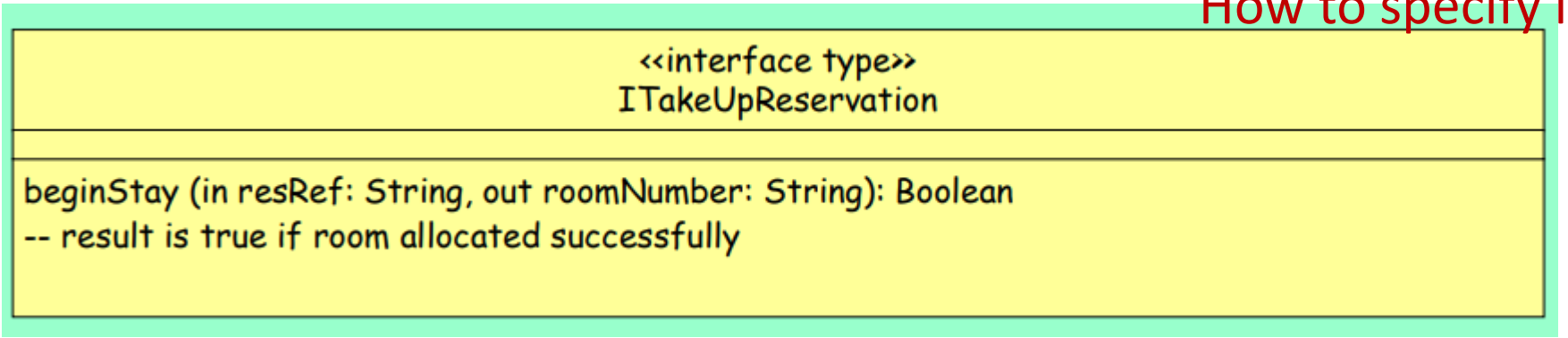

/IBilling
beginStay( )
/ITakeUpReservation:ReservationSystem
/ICustomerMgt
/IHotelMgt

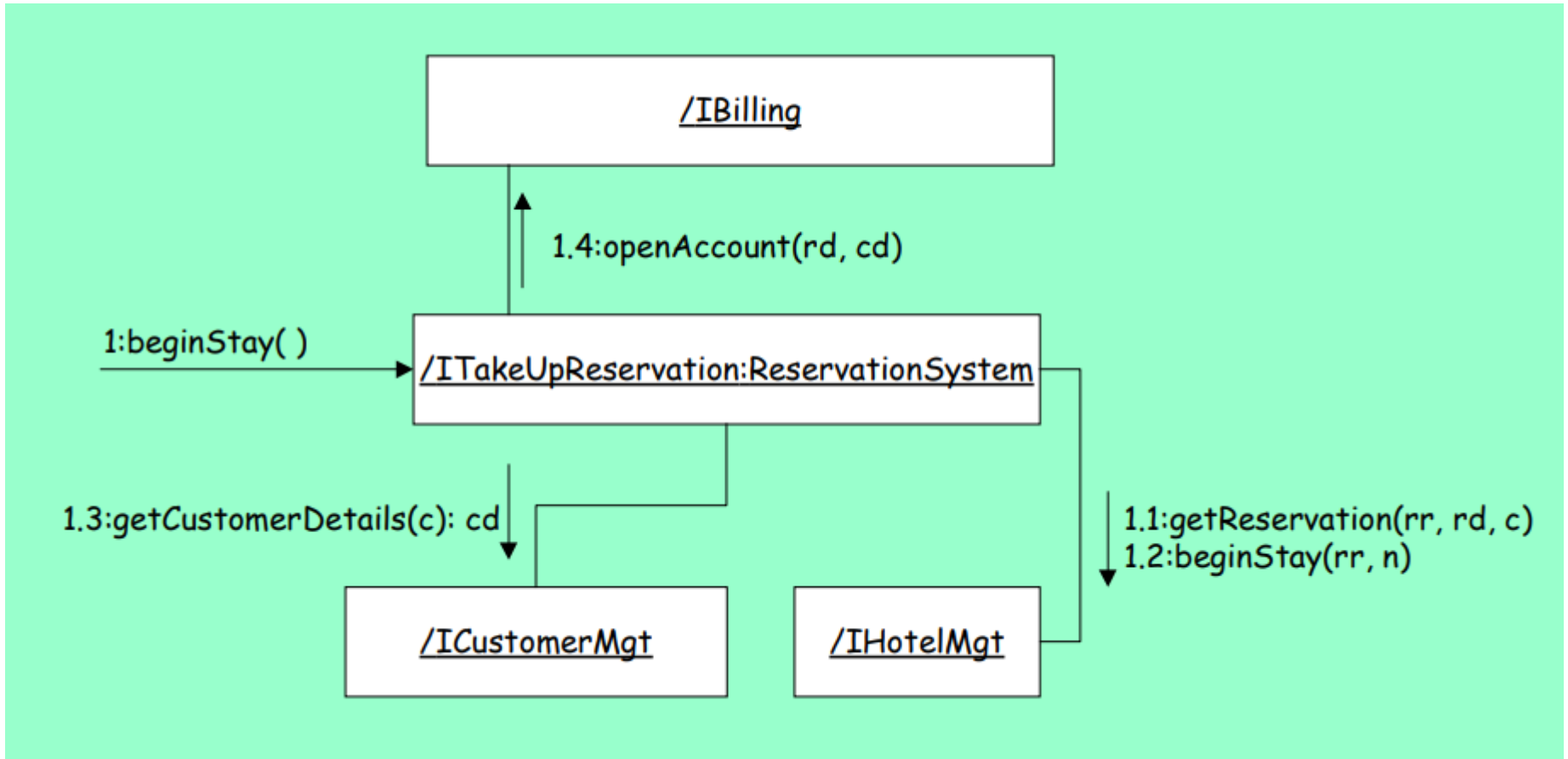

<<interface type>>
ITakeUpReservation
beginStay (in resRef: String, out roomNumber: String): Boolean
-- result is true if room allocated successfully


<<interface type>>
IHotelMgt
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation (in res: ReservationDetails, in cus: CustId, out resRef: String): Boolean
getReservation(in resRef: String, out rd ReservationDetails, out cusId: CustId): Boolean
beginStay (resRef: String , out roomNumber: String): Boolean
<<interface type>>
ICustomerMgt
getCustomerMatching (in custD: CustomerDetails, out cusId: CustId): Integer
createCustomer(in custD: CustomerDetails, out cusId: CustId): Boolean
notifyCustomer (in cus: CustId, in msg: String)
getCustomerDetails (in cus: CustId): CustomerDetails
<<interface type>>
IBilling
openAccount (in res: ReservationDetails, in cus: CustomerDetails)



Exercise:
How to specify interaction?

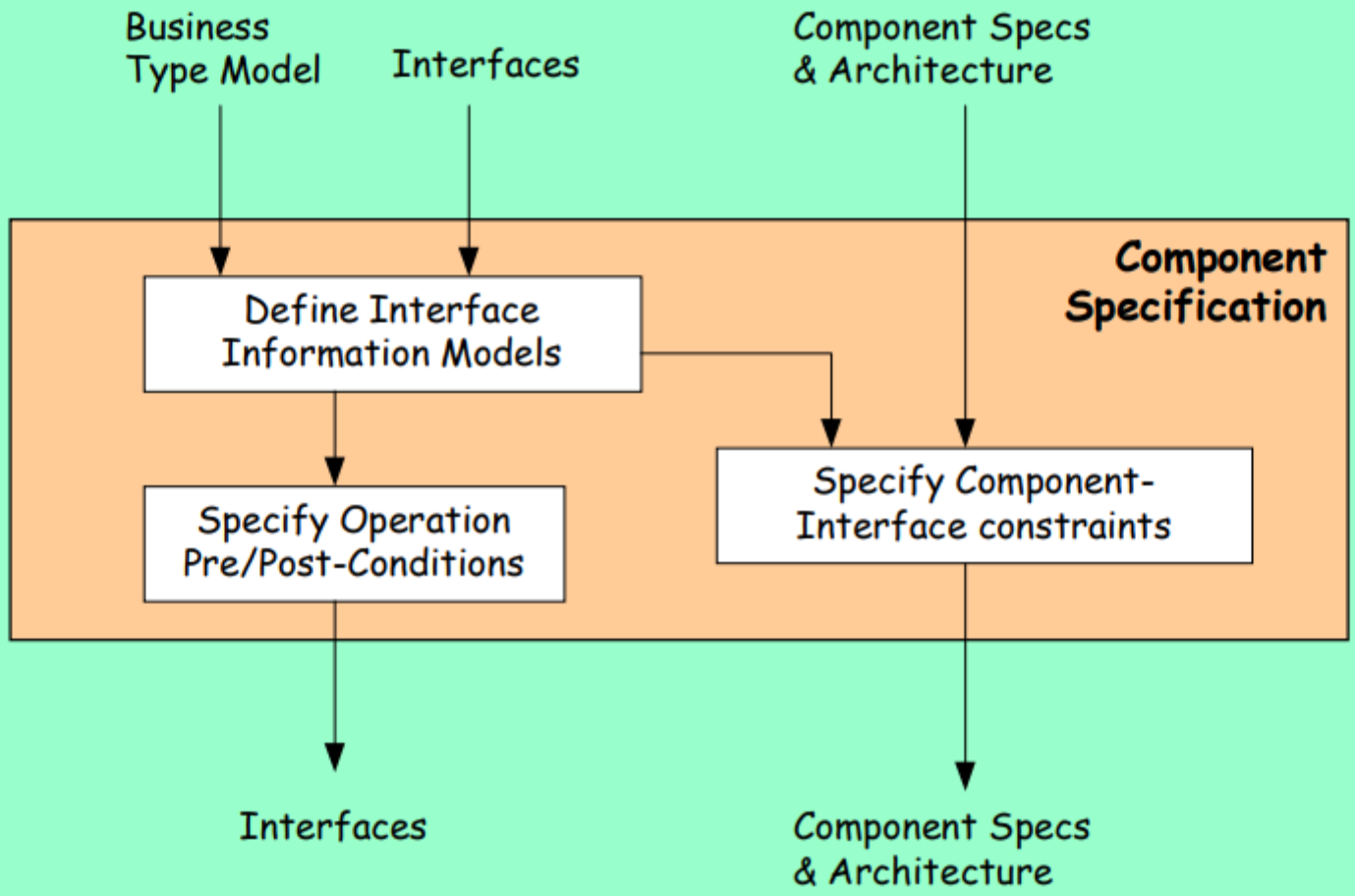



/IBilling
1.4:openAccount(rd, cd)
1:beginStay( )
/ITakeUpReservation:ReservationSystem
1.3:getCustomerDetails(c): cd
1.1:getReservation(rr, rd, c)
1.2:beginStay(rr, n)
/ICustomerMgt
/IHotelMgt

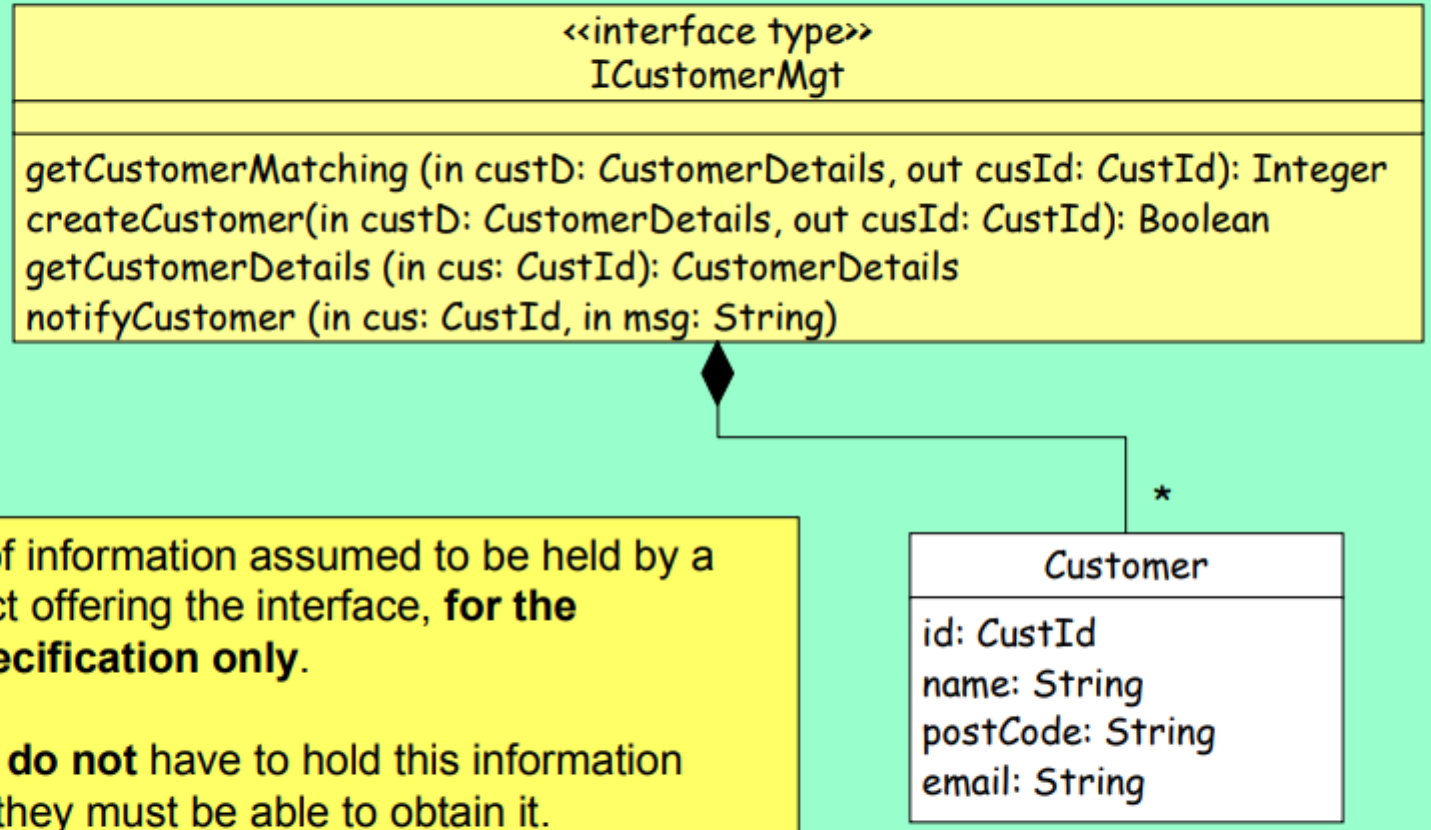
Tutorial Map

- Requirements Definition
- Component Identification
- Component Interaction
- Component Specification

Component Specification



Interface information model



Defines the set of information assumed to be held by a component object offering the interface, **for the purposes of specification only**.

Implementations **do not** have to hold this information themselves, but they must be able to obtain it.

The model need only be sufficient to explain the effects of the operations.

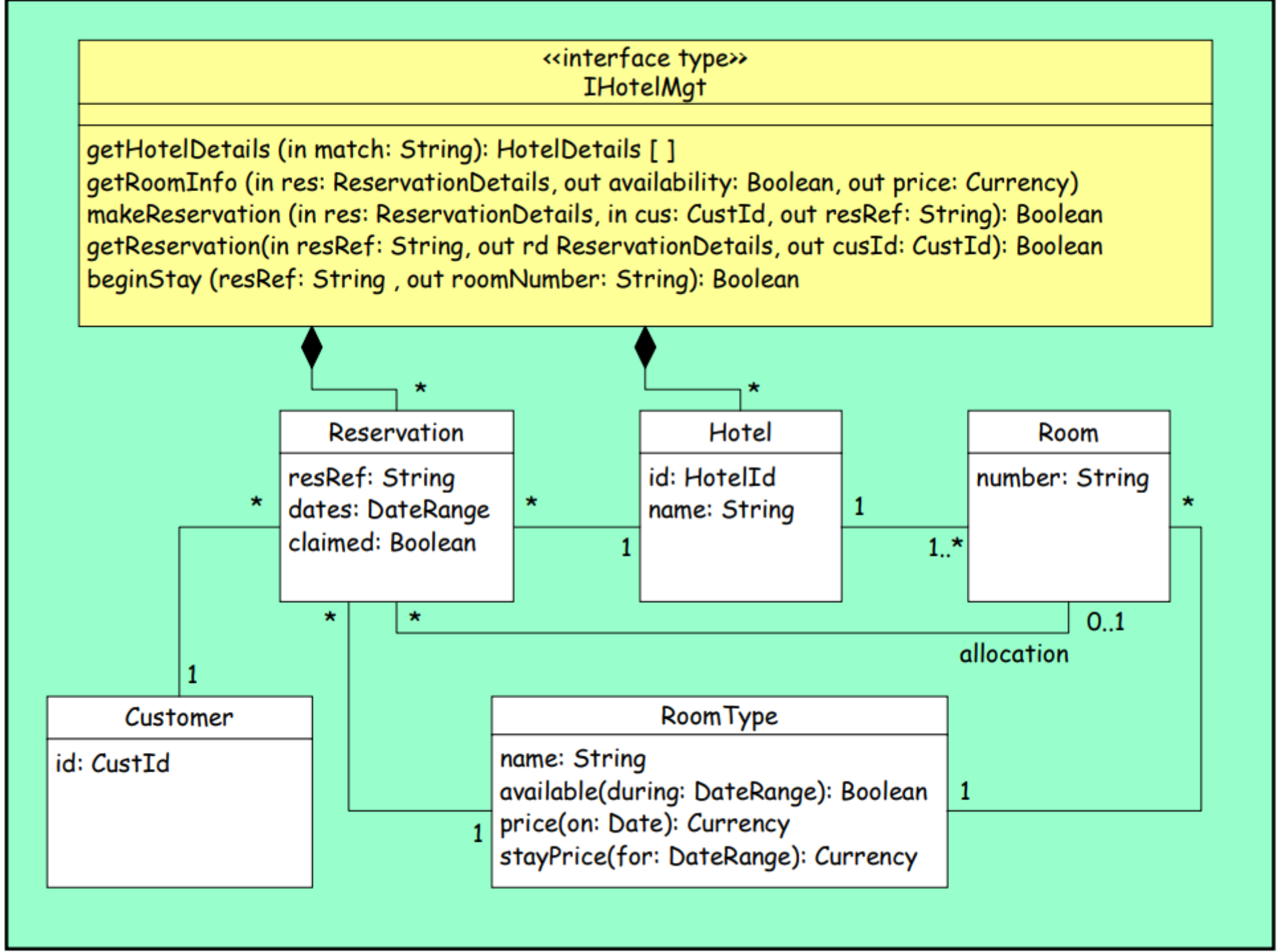
The model can be derived from the Business Type Model.

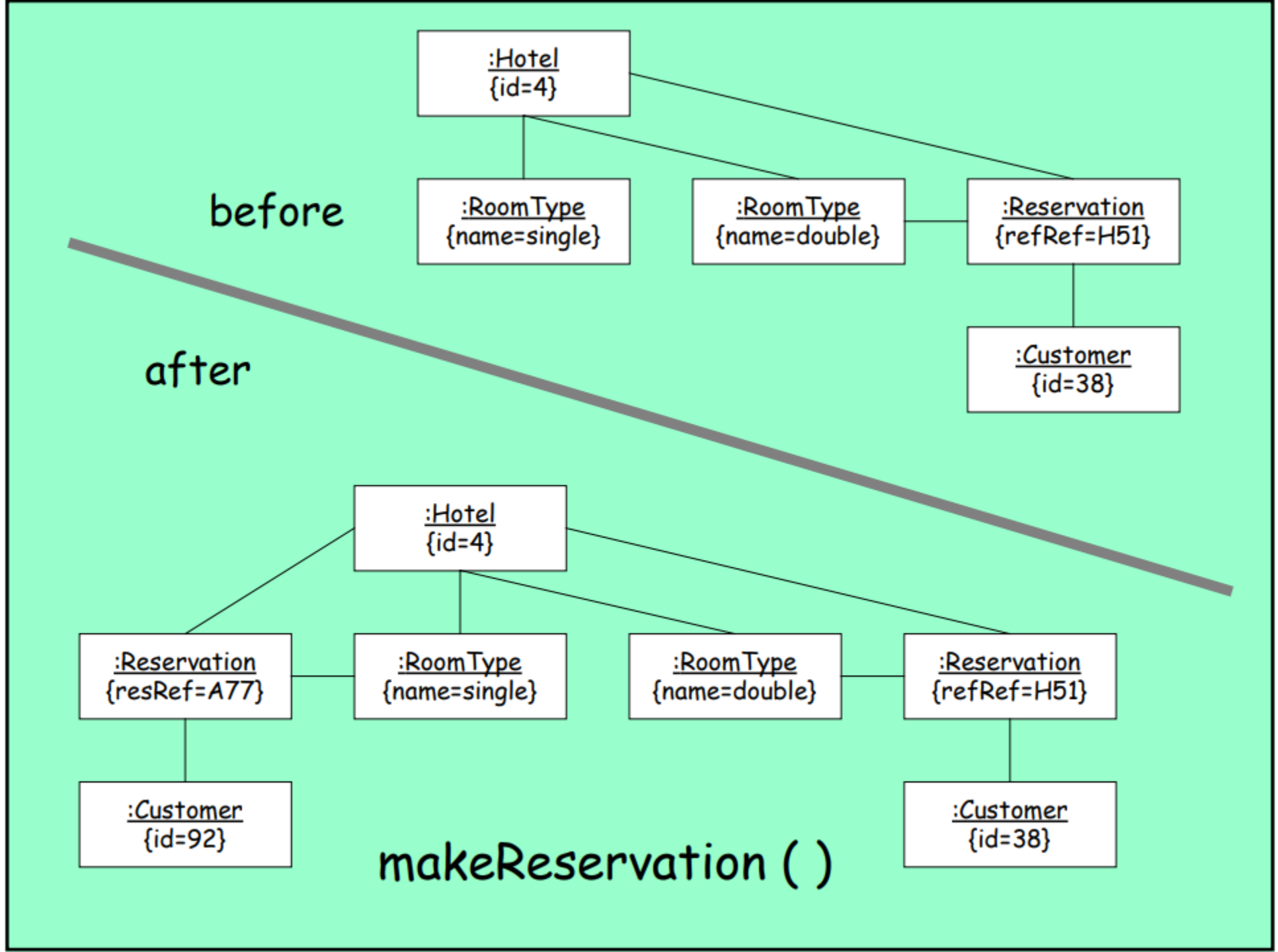
Pre- and post-conditions

- If the pre-condition is true, the post-condition must be true
- If the pre-condition is false, the post-condition doesn't apply
- A missing pre-condition is assumed 'true'
- Pre- and post-conditions can be written in natural language or in a formal language such as OCL

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

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

post:
    -- the details returned match those held for customer cus
    Let theCust = customer->select(c | c.id = cus) in
    result.name = theCust.name
    result.postCode = theCust.postCode
    result.email = theCust.email
```

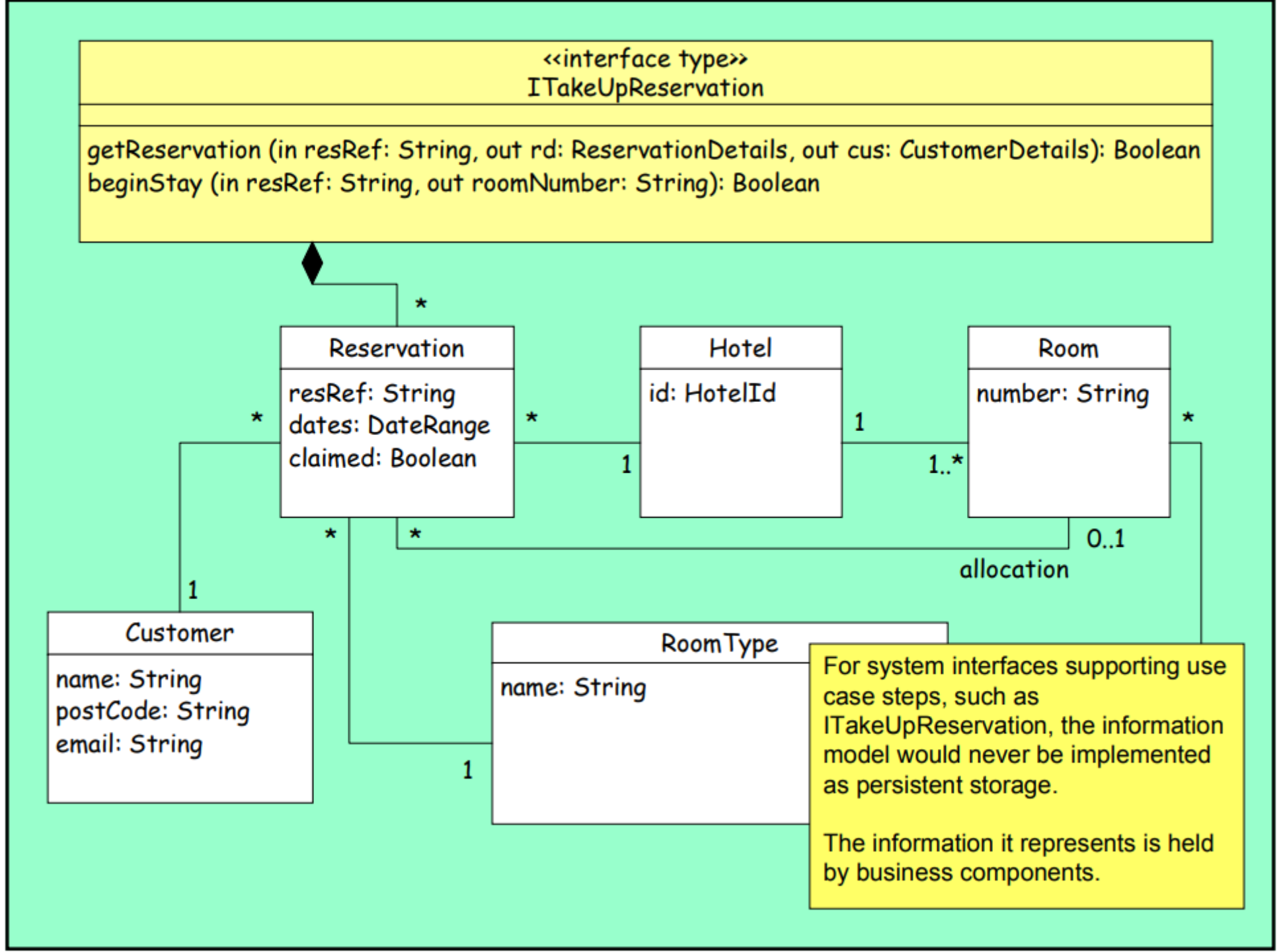
```
context ICustomerMgt::createCustomer (in custD: CustomerDetails, out cusId: CustId): Boolean

pre:
      -- post code and email address must be provided
      custD.postCode->notEmpty and custD.email->notEmpty

post:
      result implies
            -- new customer (with name not previously known) created
            (not customer@pre->exists(c | c.name = custD.name)) and
            (customer - customer@pre)->size = 1 and
                  Let c = (customer - customer@pre) in
                        c.name = custD.name and c.postCode = custD.postCode and
                        c.email = custD.email and c.id = cusId
```


«interface type»
IHotelMgt
getHotelDetails (in match: String): HotelDetails [ ]
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation (in res: ReservationDetails, in cus: CustId, out resRef: String): Boolean
getReservation(in resRef: String, out rd ReservationDetails, out cusId: CustId): Boolean
beginStay (resRef: String , out roomNumber: String): Boolean
Reservation
resRef: String
dates: DateRange
claimed: Boolean
Hotel
id: HotelId
name: String
Room
number: String
Customer
id: CustId
RoomType
name: String
available(during: DateRange): Boolean
price(on: Date): Currency
stayPrice(for: DateRange): Currency
allocation
0..1
1..*
*
1

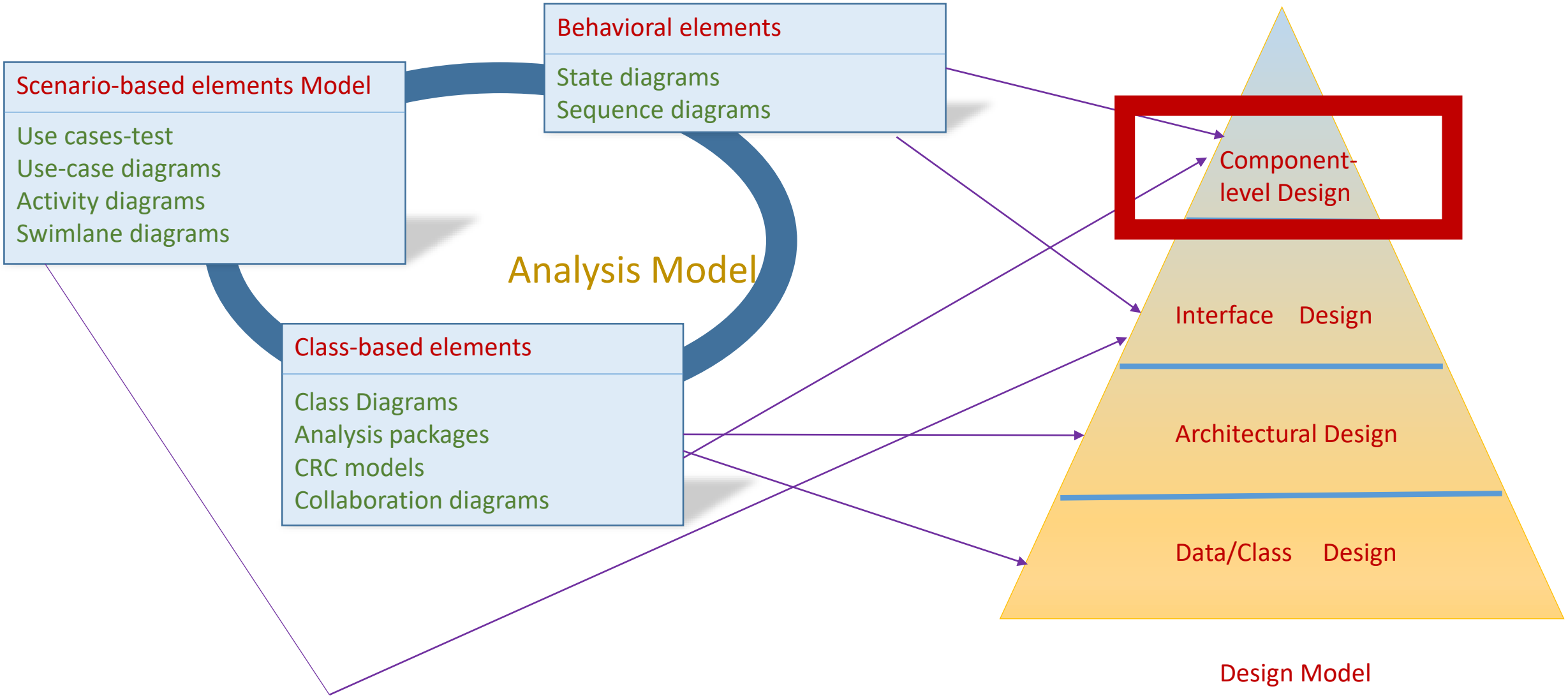

:Hotel
{id=4}
before
:RoomType
{name=single}
:RoomType
{name=double}
:Reservation
{refRef=H51}
:Customer
{id=38}
after
:Hotel
{id=4}
:Reservation
{resRef=A77}
:RoomType
{name=single}
:RoomType
{name=double}
:Reservation
{refRef=H51}
:Customer
{id=92}
:Customer
{id=38}
makeReservation ( )

```
context makeReservation (in res: ReservationDetails, in cus: CustId, out 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
        Let h = hotel->select(x | x.id = res.hotel)->asSequence->first in
            -- only one more reservation now than before
            (h.reservation - h.reservation@pre)->size = 1 and
            -- identify the reservation
            Let r = (h.reservation - h.reservation@pre)->asSequence->first in
                -- return number is number of the new reservation
                r.resRef = resRef and
                -- other attributes match
                r.dates = res.dateRange and
                r.roomType.name = res.roomType and not r.claimed and
                r.customer.id = cus
```

```
context IHotelMgt::beginStay (in resRef: String, out roomNumber: String): Boolean

pre:
      -- resRef is valid
      reservation->exists (r | r.resRef = resRef) and
      -- not already claimed
      not reservation->exists (r | r.resRef = resRef and r.claimed)

post:
      Let res = reservation->select (r | r.resRef = resRef) in
      result implies
            -- the reservation is now claimed
            res.claimed and
            roomNumber = res.allocation.number
            -- nb room allocation policy not defined
```


<<interface type>>
ITakeUpReservation
getReservation (in resRef: String, out rd: ReservationDetails, out cus: CustomerDetails): Boolean
beginStay (in resRef: String, out roomNumber: String): Boolean
Reservation
resRef: String
dates: DateRange
claimed: Boolean
Hotel
id: HotelId
Room
number: String
Customer
name: String
postCode: String
email: String
RoomType
name: String
allocation
0..1
1..*
For system interfaces supporting use case steps, such as ITakeUpReservation, the information model would never be implemented as persistent storage.
The information it represents is held by business components.

```
context ITakeUpReservation::beginStay (in resRef: String, out roomNumber: String): Boolean

pre:
    -- resRef is valid
    reservation->exists (r | r.resRef = resRef) and
    -- not already claimed
    not reservation->exists (r | r.resRef = resRef and r.claimed)

post:
    Let res = reservation->select (r | r.resRef = resRef) in
    result implies
        -- the reservation is now claimed
        res.claimed and
        roomNumber = res.allocation.number
        -- nb room allocation policy not defined here
```


Behavioral elements
State diagrams
Sequence diagrams
Scenario-based elements Model
Use cases-test
Use-case diagrams
Activity diagrams
Swimlane diagrams
Analysis Model
Class-based elements
Class Diagrams
Analysis packages
CRC models
Collaboration diagrams
Component-level Design
Interface Design
Architectural Design
Data/Class Design
Design Model


ANY
QUESTIONS?