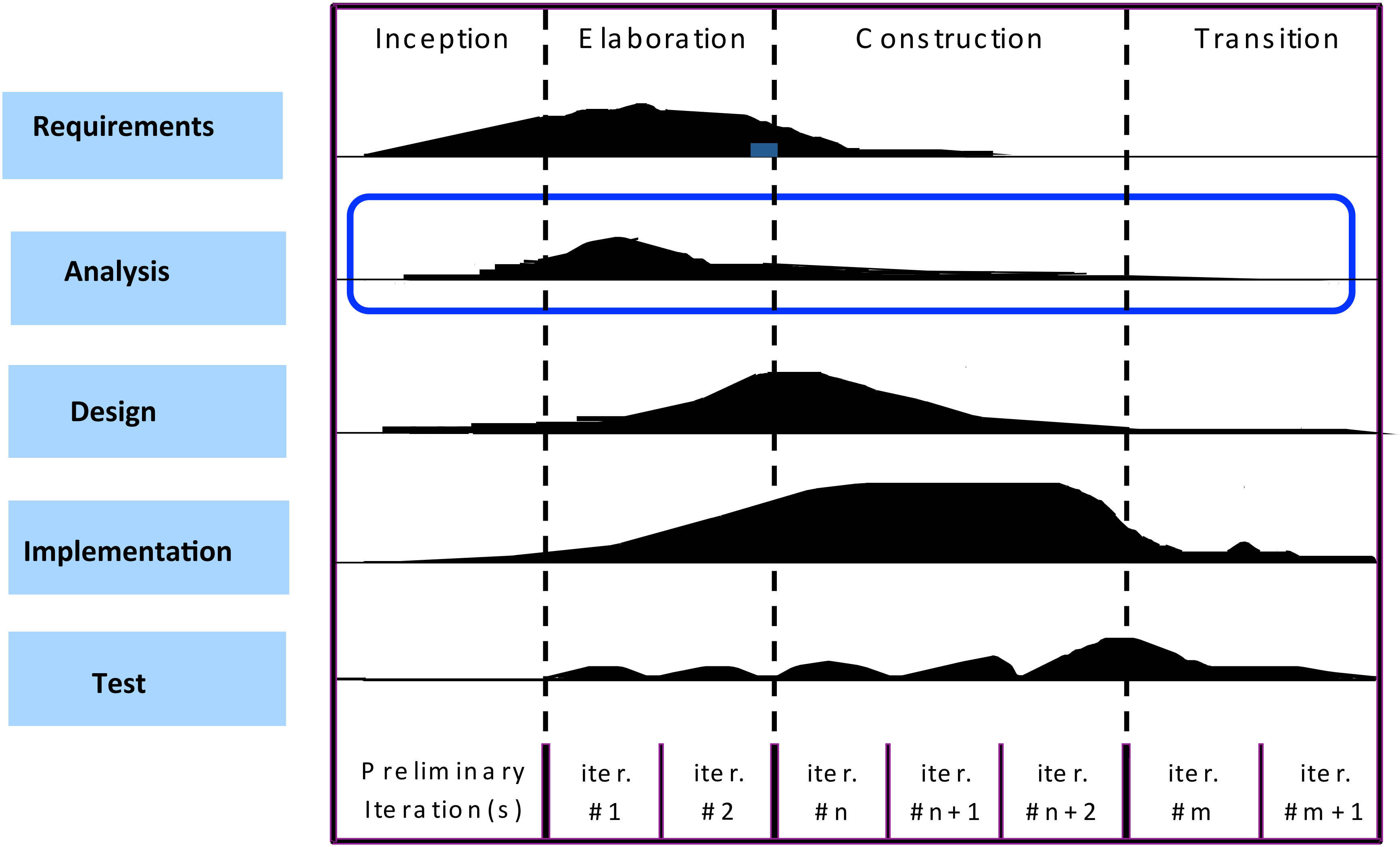
# Software Design
# Models, Tools & Processes

## Lecture 3: Elaboration Phase

Cecilia Mascolo

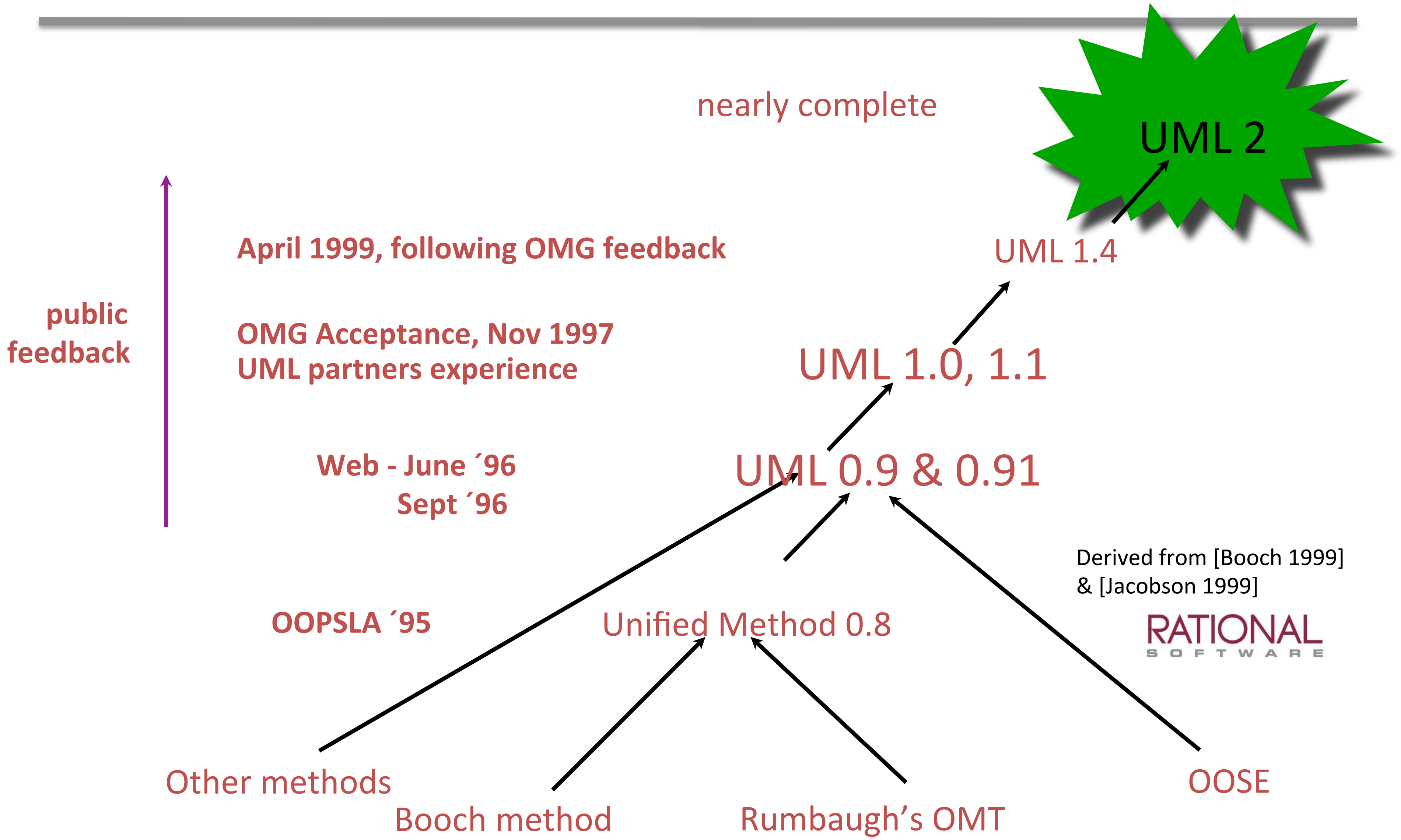
# USDP context



# Pioneers – Peter Chen

- Entity-Relationship Modeling
  - 1976, Massachusetts Institute of Technology
- User-oriented response to Codd's relational database model
  - Define attributes and values
  - Relations as associations between things
  - Things play a *role* in the relation.
- E-R Diagrams showed entity (box), relation (diamond), role (links).
- Object-oriented Class Diagrams show class (box) and association (links)

# UML history & status

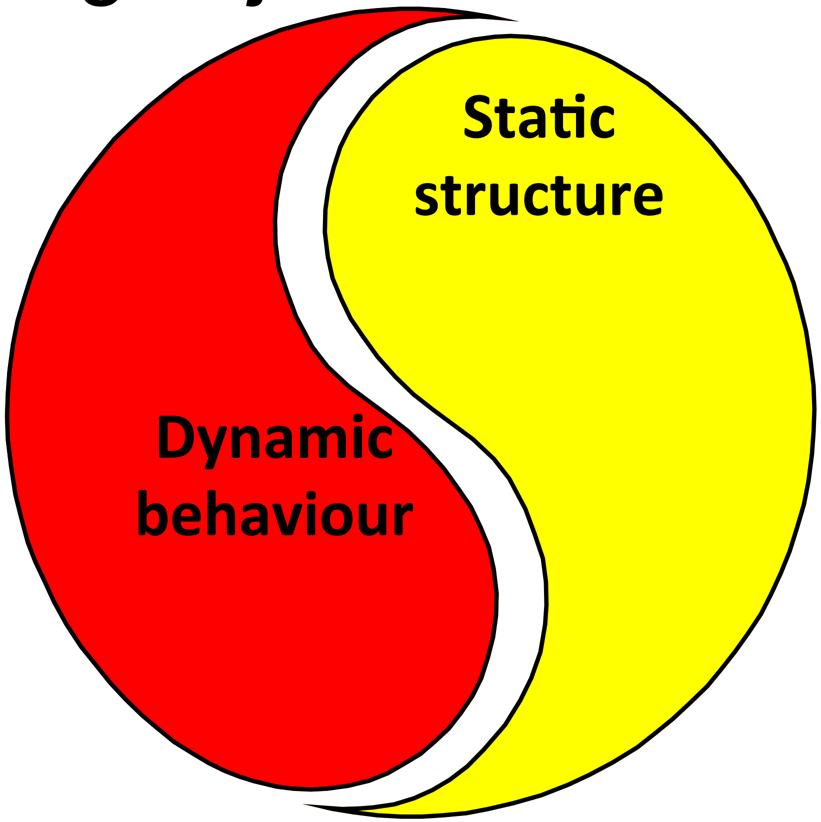


# Review of objects and classes

- objects
  - represent ‘things’ in some problem domain (example: “the red car down in the car park”)
- classes
  - represent all objects of a kind (example: “car”)
- operations
  - actions invoked on objects (Java “*methods*”)
- instance
  - can create many instances from a single class
- state
  - all the attributes (field values) of an instance

# Premise

- It is possible to model a software system (or other system) as a collection of collaborating objects



# Modelling elements

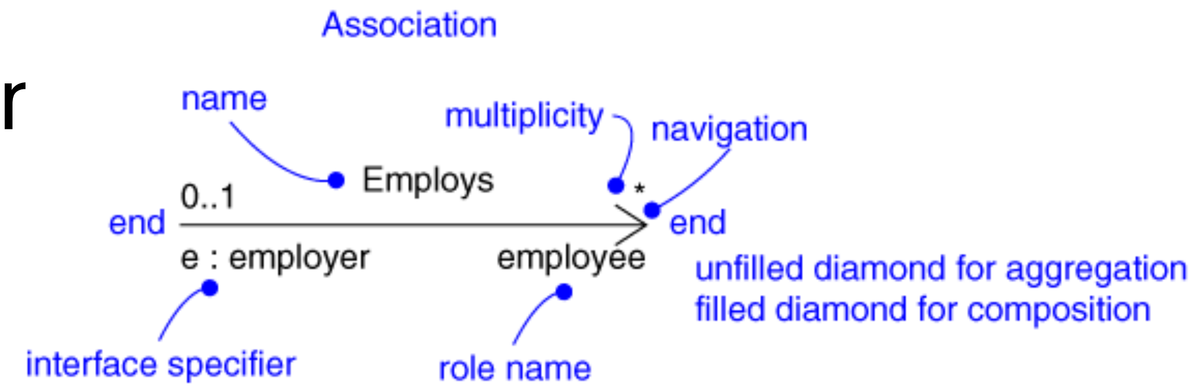
- Structural elements
  - Class, interface, collaboration, use case, active class, component, node
- Behavioral elements
  - Interaction, state machine
- Grouping elements
  - Package, subsystem

  - Capture the *requirements* of a system
- Other elements
  - Note

Adopted from [Booch 1999]

# Relationships

- Dependency
- Association
- Generalisatior
- Realisation



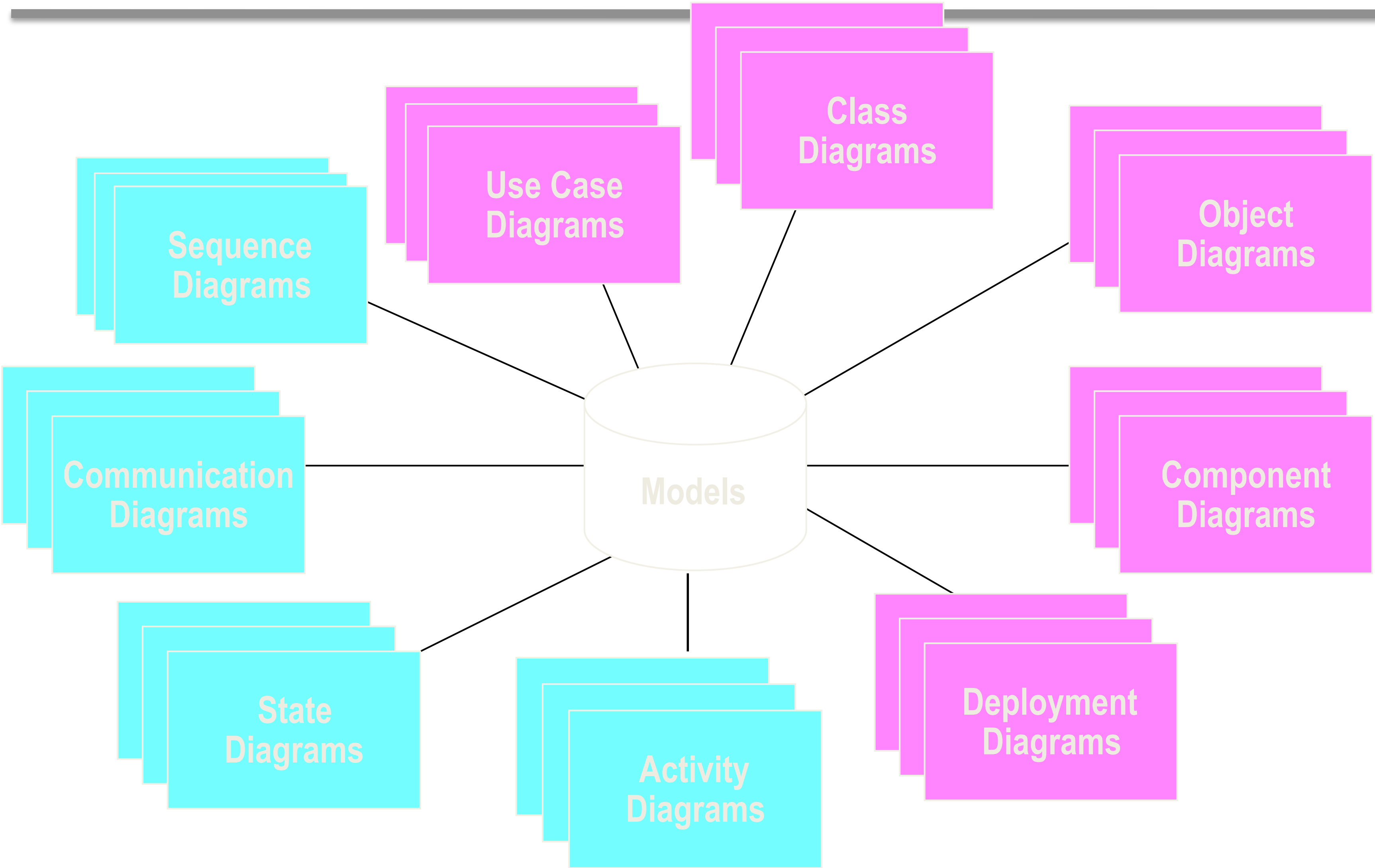

Taken from [Booch 1999]

# Diagrams

- A *diagram* is a view into a model
  - Presented from the aspect of a particular stakeholder
  - Provides a partial representation of the system
  - Is semantically consistent with other views
- In UML, there are nine standard diagrams
  - **Static views:** use case, class, object, component, deployment
  - **Dynamic views:** sequence, collaboration, statechart, activity

Taken from [Booch 1999]

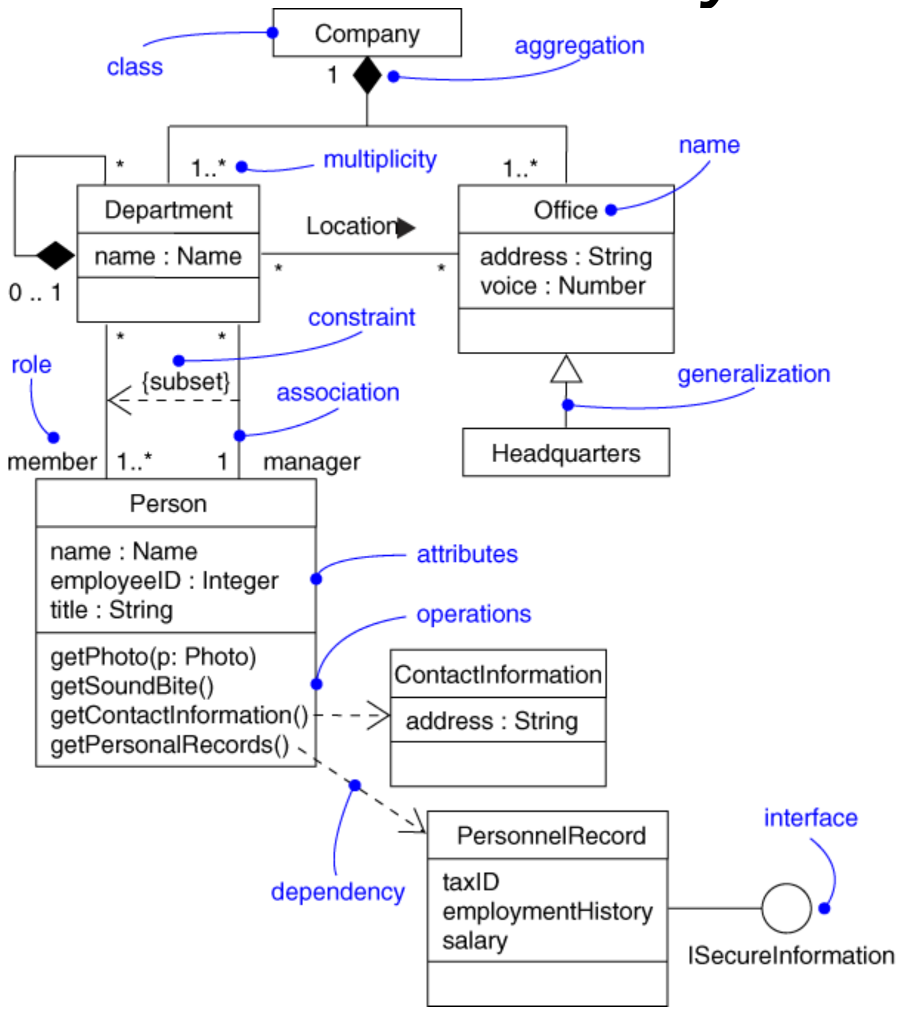
# UML models, views & diagra




Taken from [Booch 1999]

# Class diagram

- Captures the *vocabulary* of a system




Taken from [Booch 1999]

# Class diagram (cont…)

- Built & refined throughout development
- Purpose
  - Name & model *concepts* in the system
  - Specify collaborations
- Developed by analysts, designers & implementers

Taken from [Booch 1999]

# Deriving objects from a scenario

- The ***nouns*** in a description refer to 'things'.
  - A source of classes and objects.
- The ***verbs*** refer to actions.
  - A source of interactions between objects.
  - Actions describe object behavior, and hence required methods.

# Example of context description

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers’ telephone number.

# Nouns

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

# Verbs

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

# Extracted nouns & verbs

**Cinema booking system**
Stores (seat bookings)
Stores (telephone number)

**Theatre**
Has (seats)

**Movie**

**Customer**
Reserves (seats)
Is given (row number, seat number)
Requests (seat booking)

**Time**

**Date**

**Seat booking**

**Show**
Is scheduled (in theatre)

**Seat**

**Seat number**

**Telephone number**

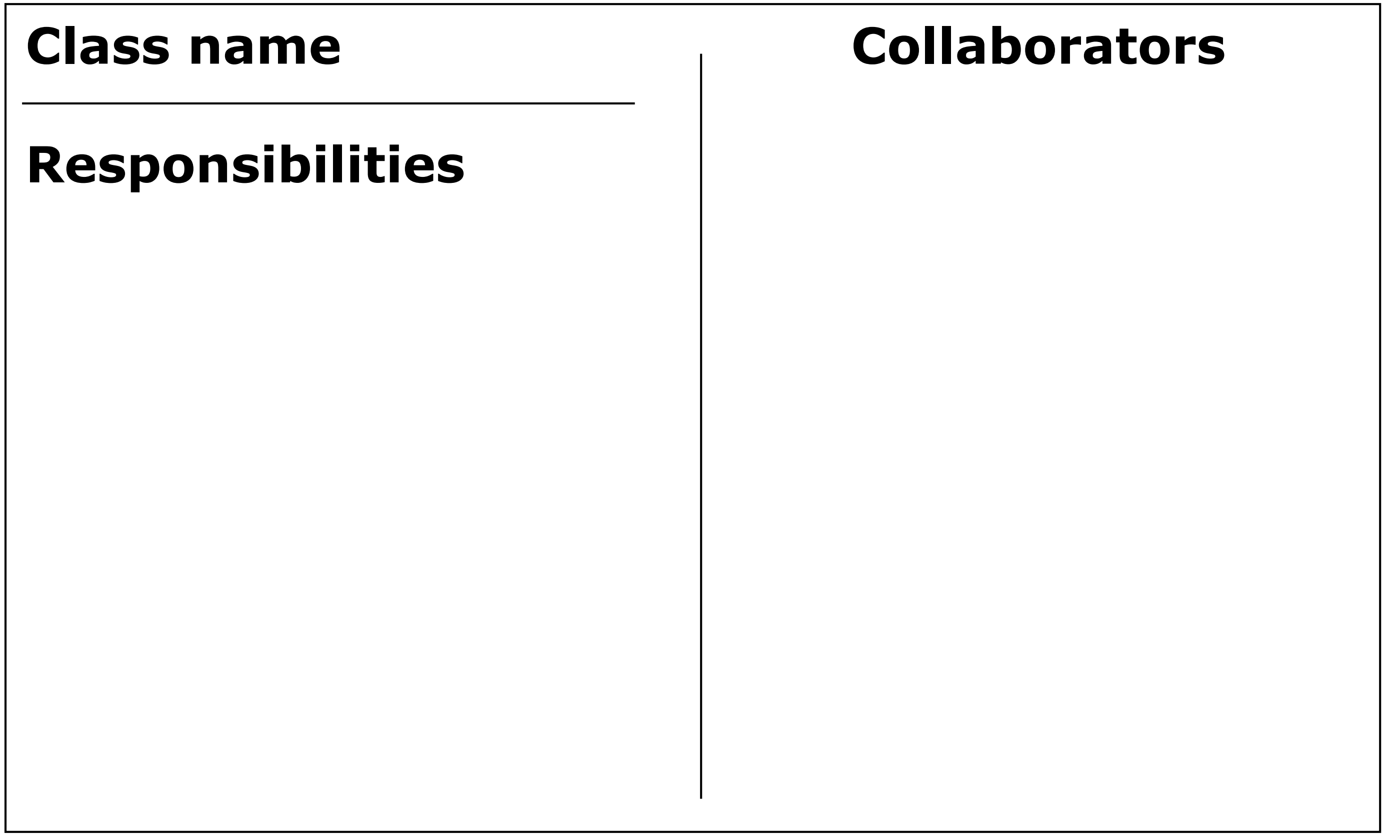
**Row**

**Row number**

# Scenario structure: CRC cards

- First described by Kent Beck and Ward Cunningham.
  - Later innovators of “agile” programming, and also the first wiki!
- Use simple index cards, with each cards recording:
  - A *class* name.
  - The class’s *responsibilities*.
  - The class’s *collaborators*.

# Typical CRC card



# Partial example

| CinemaBookingSystem | *Collaborators* |
| --- | --- |
| Can find movies by title and day.<br>Stores collection of movies.<br>Retrieves and displays movie details.<br>... | Movie<br><br>Collection |

# Dividing up a design model

- Abstraction
  - Ignore details in order to focus on higher level problems (e.g. aggregation, inheritance).
  - If classes correspond well to types in domain they will be easy to understand, maintain and reuse.
- Modularization
  - Divide model into parts that can be built and tested separately, interacting in well-defined ways.
  - Allows different teams to work on each part
  - Clearly defined interfaces mean teams can work independently & concurrently, with increased chance of successful integration.

# Class design from CRC cards

- Scenario analysis helps to clarify application structure.
  - Each card maps to a class.
  - Collaborations reveal class cooperation/object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g. “Stores collection ...”

# Refining class interfaces

- Replay the scenarios in terms of method calls, parameters and return values.
- Note down the resulting method signatures.
- Create outline classes with public-method stubs.
- Careful design is a key to successful implementation.

# Dividing up a design model

- Abstraction
  - Ignore details in order to focus on higher level problems (e.g. aggregation, inheritance).
  - If classes correspond well to types in domain they will be easy to understand, maintain and reuse.
- Modularization
  - Divide model into parts that can be built and tested separately, interacting in well-defined ways.
  - Allows different teams to work on each part
  - Clearly defined interfaces mean teams can work independently & concurrently, with increased chance of successful integration.

# Pioneers – David Parnas

- Information Hiding
  - 1972, Carnegie Mellon University
- How do you decide the points at which a program should be split into pieces?
  - Are small modules better?
  - Are big modules better?
  - What is the optimum boundary size?
- Parnas proposed the best criterion for modularization:
  - Aim to hide design decisions within the module.

# Information hiding in OO models

- Data belonging to one object is hidden from other objects.
  - Know *what* an object can do, not *how* it does it.
  - Increases independence, essential for large systems and later maintenance
- Use Java visibility to hide implementation
  - Only methods intended for interface to other classes should be public.
  - Fields should be private – accessible only within the same class.
  - *Accessor* methods provide information about object state, but don't change it.
  - *Mutator* methods change an object's state.

# Cohesion in OO models

- Aim for high cohesion:
  - Each component achieves only “one thing”
- Method (functional) cohesion
  - Method only performs out one operation
  - Groups things that must be done together
- Class (type) cohesion
  - Easy to understand & reuse as a domain concept
- Causes of low, poor, cohesion
  - Sequence of operations with no necessary relation
  - Unrelated operations selected by control flags
  - No relation at all – just a bag of code

# Summary

- We have described the main activity of the elaboration phase
- We have introduced class diagrams as well as CRC cards and the process of identifying relevant classes.