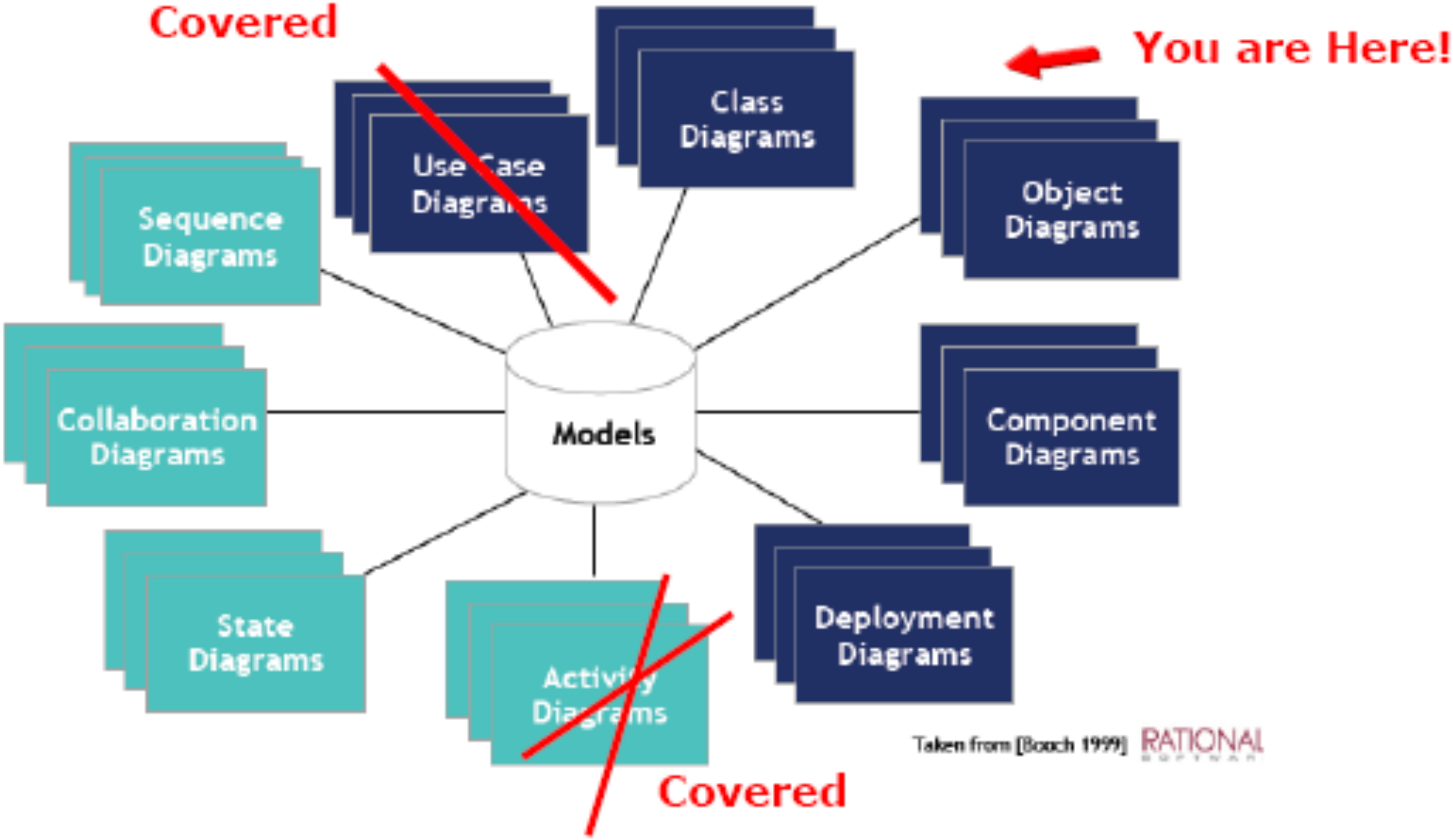
UML Diagrams



Taken from [Booch 1999]

Class diagrams

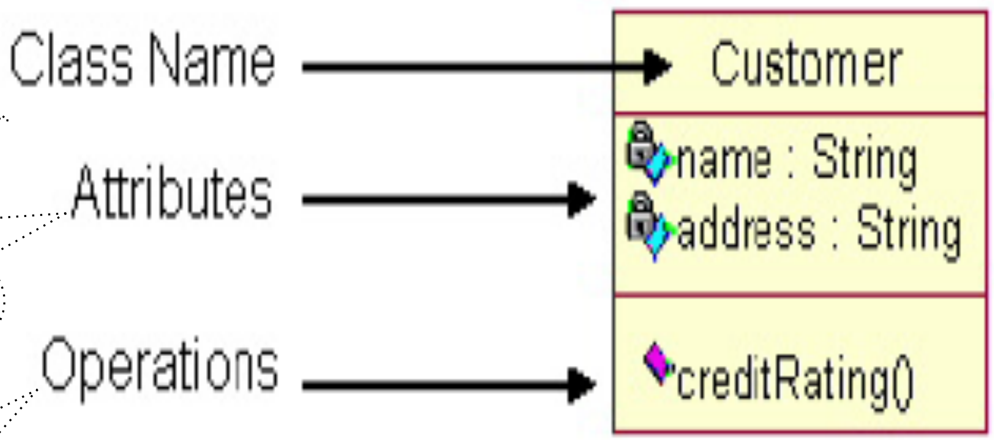
Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.

An object class can be thought of as a general definition of one kind of system object.

An association is a link between classes that indicates that there is some relationship between these classes.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

Simple Class Diagram

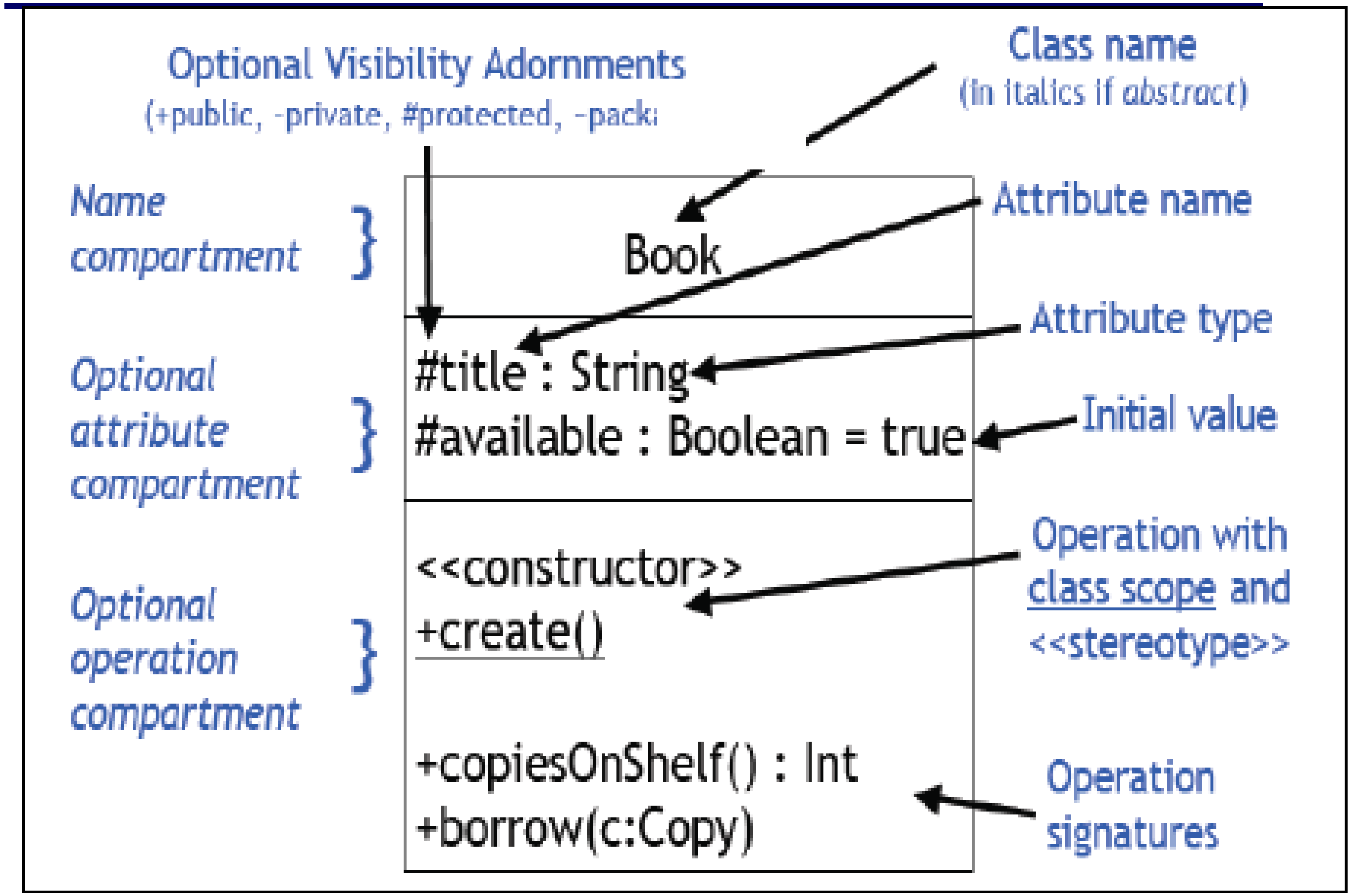


This represents a system object, which may refer to an Actor or entity of the system

These relate to the data that system must store for this object

These relate to the use cases provided by this object

UML Class Icons


Optional Visibility Adornments
(+public, -private, #protected, -pack:
Class name
(in italics if *abstract*)
Name compartment
Book
Attribute name
Attribute type
#title : String
#available : Boolean = true
Initial value
Optional attribute compartment
<<constructor>>
+create()
Operation with class scope and <<stereotype>>
Optional operation compartment
+copiesOnShelf() : Int
+borrow(c:Copy)
Operation signatures

Reference: D. Rosenblum, UCL

+, #, -

+ means public: public members can be accessed by any client of the class

# means protected: protected members can be accessed by members of the class or any subclass

- means private: private members can only be accessed by members of the same class

Analysis Class

An analysis class abstracts one or more classes and/or

subsystems in the system's design

- Focuses on handling functional requirements
- Defines responsibilities (cohesive subsets of behaviour defined by the class, e.g. use cases or services it provides to other classes)
- Defines attributes
- Expresses relationships the class is involved in

Approach: Data-Driven Design

Identify all the data in the system

Divide into classes before considering responsibilities

Common approach: **noun identification**

- Identify candidate classes by selecting all the nouns and nouns phrases in the requirements document
- Discard inappropriate candidates
  - Redundant or omnipotent entities
  - Vague entities
  - Events or operations
  - Meta-language
  - Entities outside system scope
  - Attributes

Verbs and verb phrases highlight candidate operations!

Data-Driven Design Approach

Some heuristics/hints of what kind of things are classes [Shlaer and Mellor; Booch]:

Tangible or “**real-world**” things – e.g. book, copy, course;

Roles- e.g. library member, student, director of studies,

Events- e.g. arrival, leaving, request;

Interactions- e.g. meeting, intersection

Exercise

Perform **noun-verb** analysis of a requirements document (example text from next slide);

Underline all the noun and noun phrases,

Create a list of candidate classes (in examining the discard criteria, you may also identify some candidate attributes)

Identify all verb and verb phrases

Create a list of candidate operations and assign them to classes

Noun/Verb Analysis

Books and journals:
The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

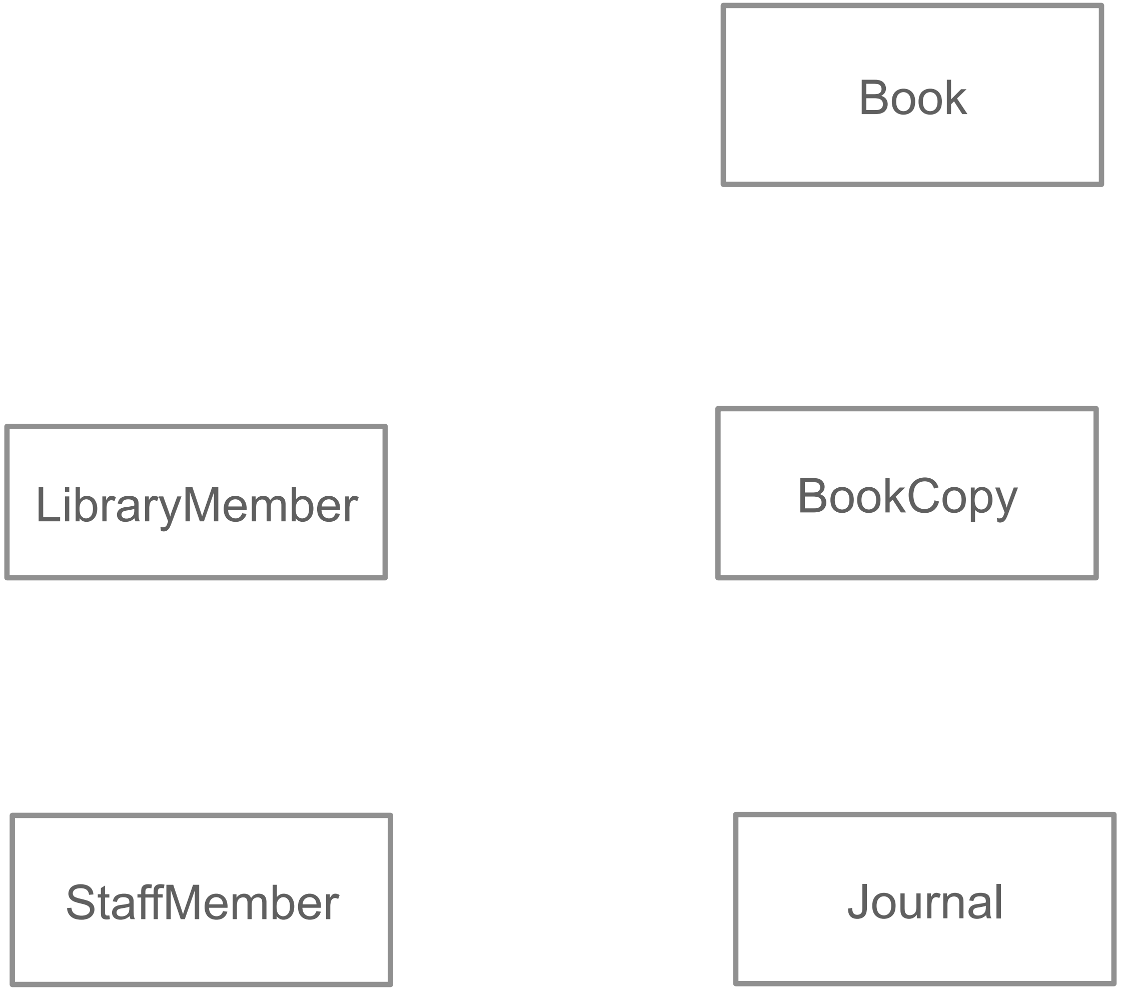
Borrowing:
The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

1. Noun Analysis

Books and journals:
The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing:
The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

First-Cut Class Diagram: Class Model (Analysis Classes)

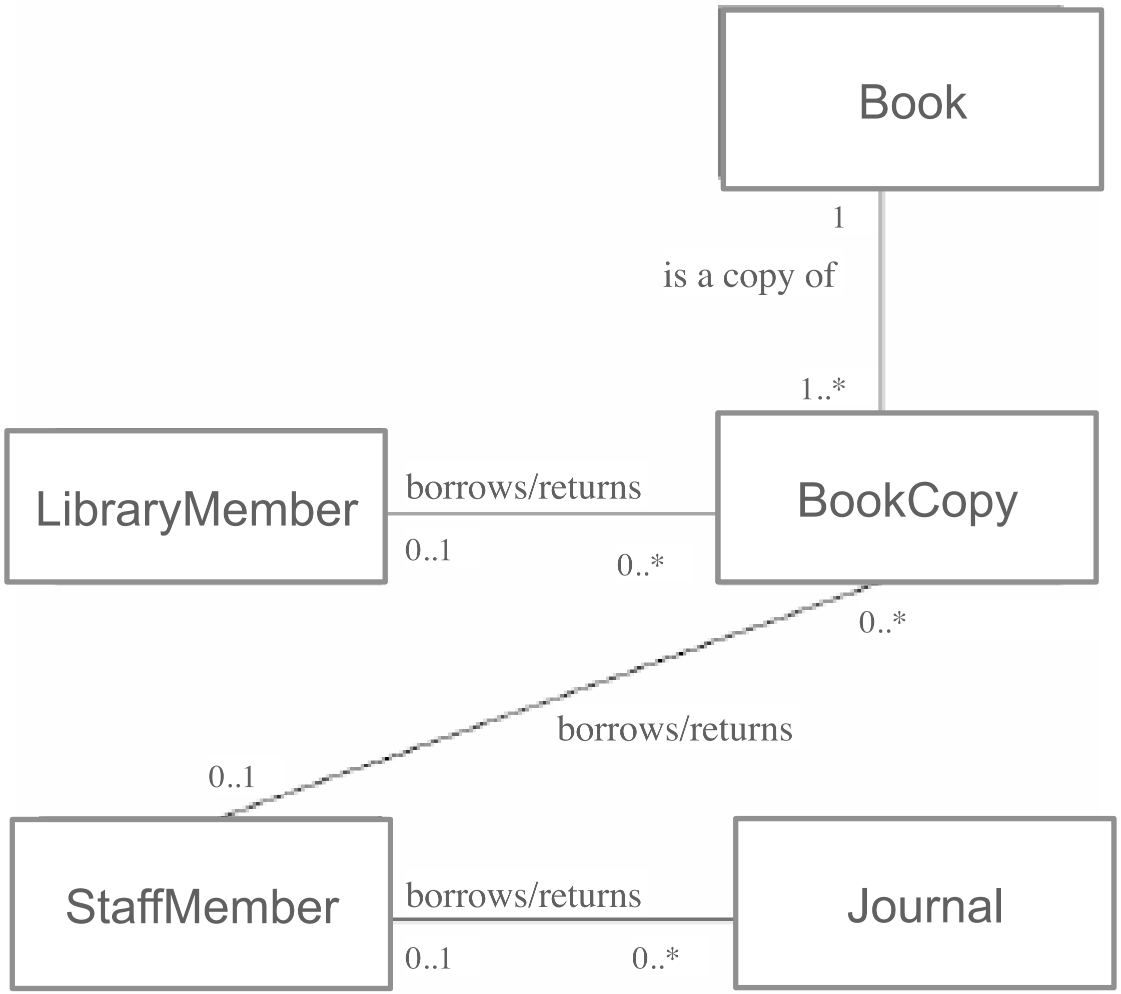


2. Verb Analysis

Books and journals:
The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing:
The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

First-Cut Class Diagram: Class Model



Book

1

is a copy of

1..*

LibraryMember

borrows/returns

0..1

0..*

BookCopy

0..*

borrows/returns

0..1

StaffMember

borrows/returns

0..1

0..*

Journal

Relationships/Associations

Relationships are connections between modelling elements

Improve understanding of the domain, describing how objects work together

Act as a sanity check for good modelling

Associations are relationships between classes

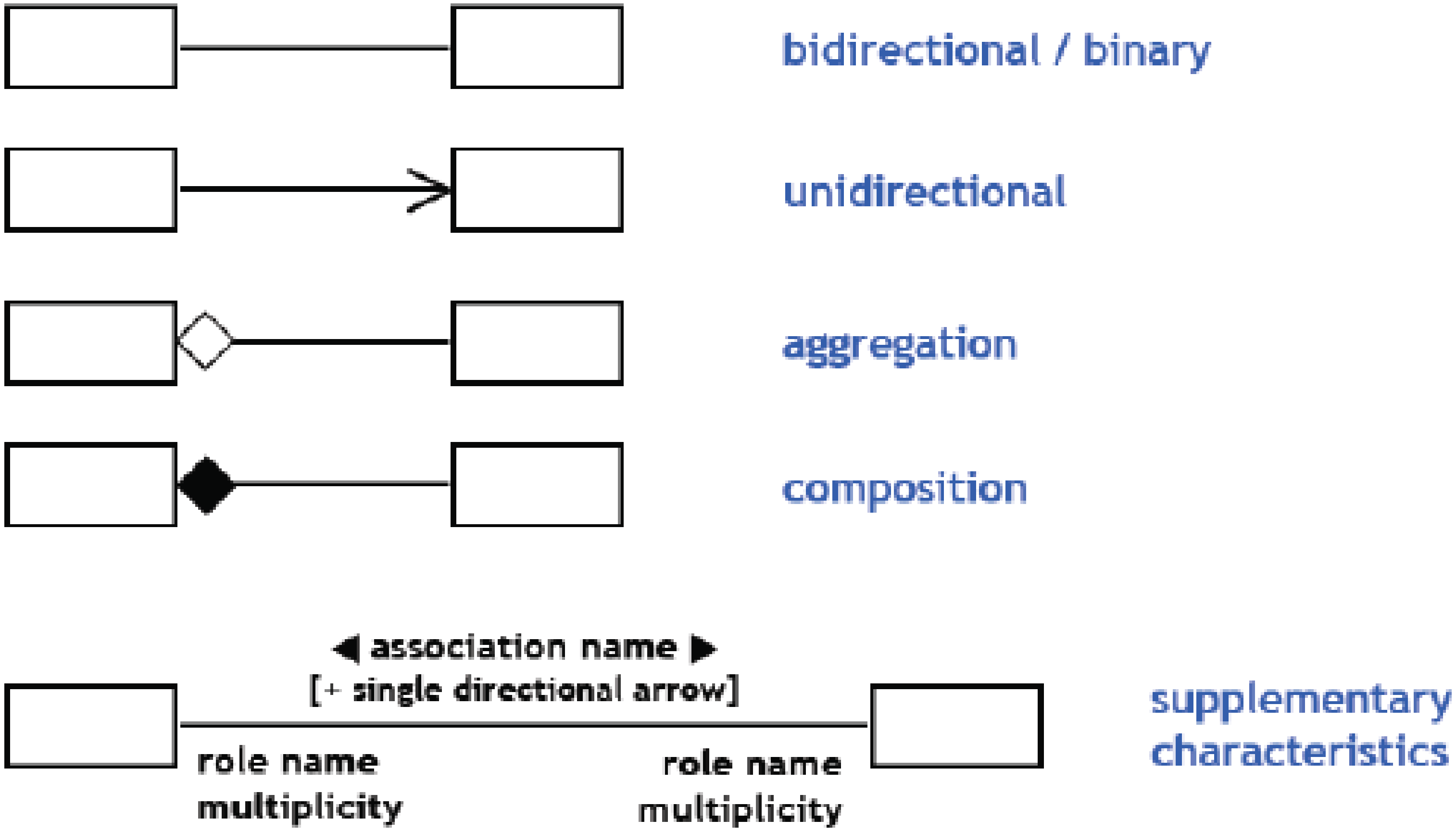
Examples

- Object of class A sends a message to object of class B
- Object of class A creates an object of class B
- Object of class A has attribute whose values are objects of class B
- Object of class A receives a message with argument of class B

Links are relationships between objects

- Links can be instances of associations (as in UML 1.4)
- Allow one object to invoke operations on another object

UML Relationships Notations



bidirectional / binary

unidirectional

aggregation

composition

◀ association name ▶
[+ single directional arrow]

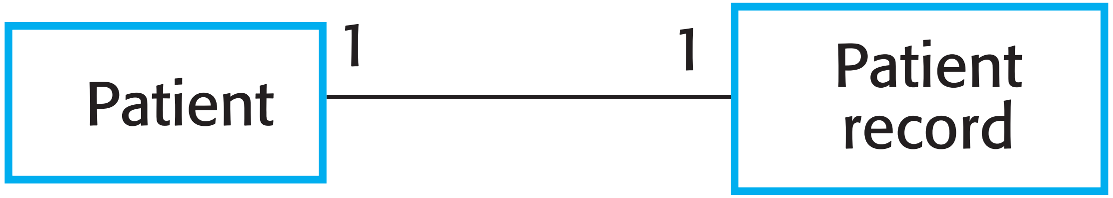
role name
multiplicity

role name
multiplicity

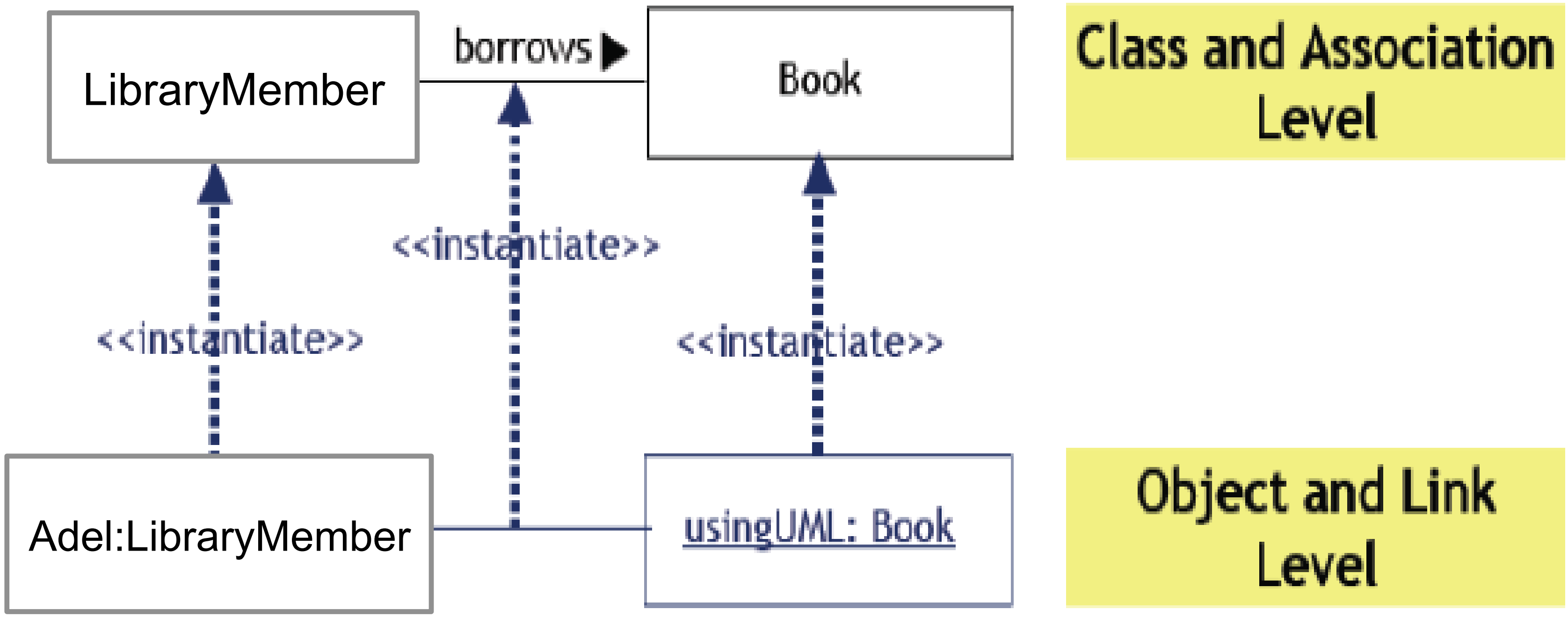
supplementary
characteristics

Reference: D. Rosenblum, UCL

UML classes and association



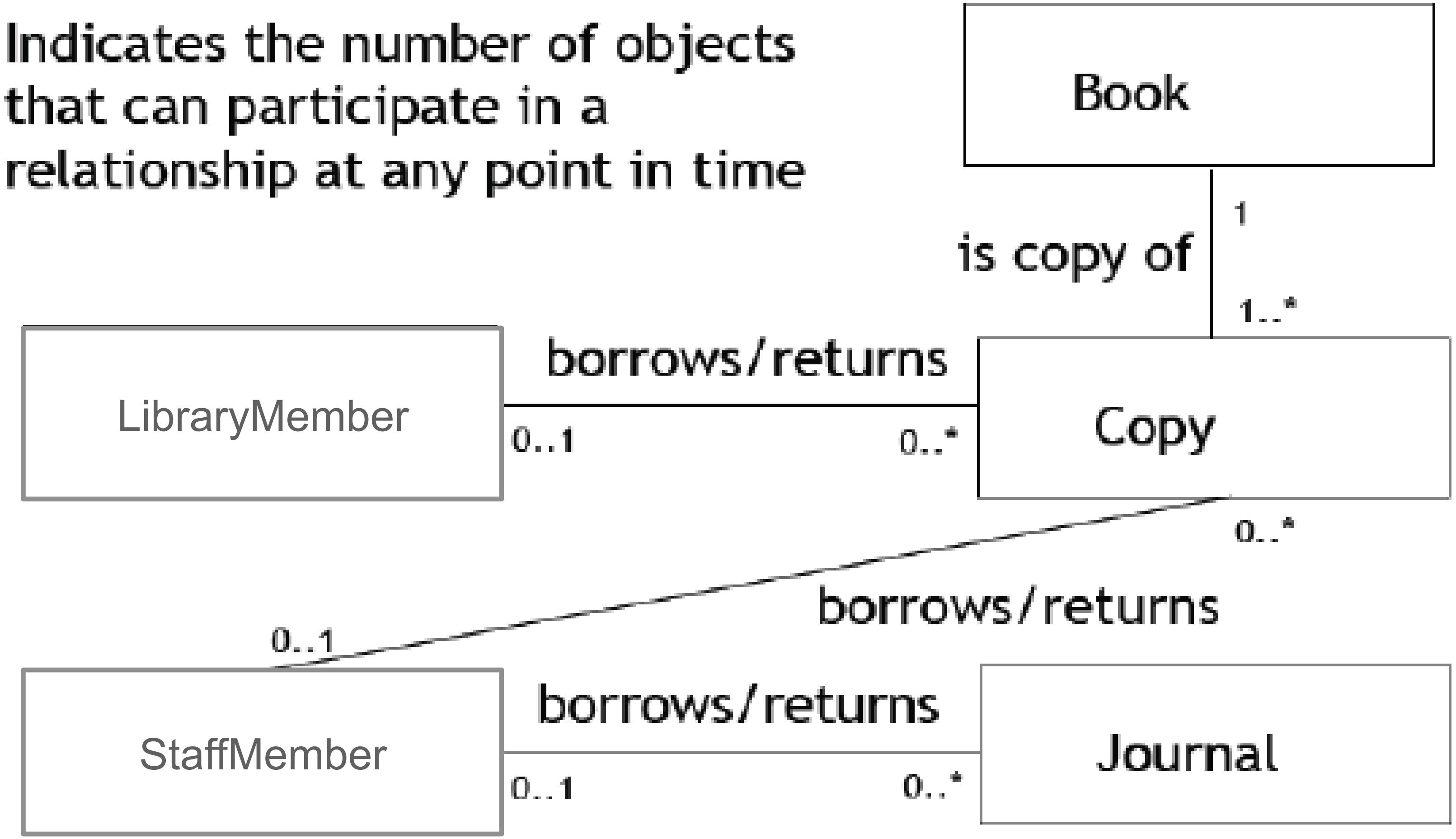
Links Instantiate Associations



Reference: D. Rosenblum, UCL

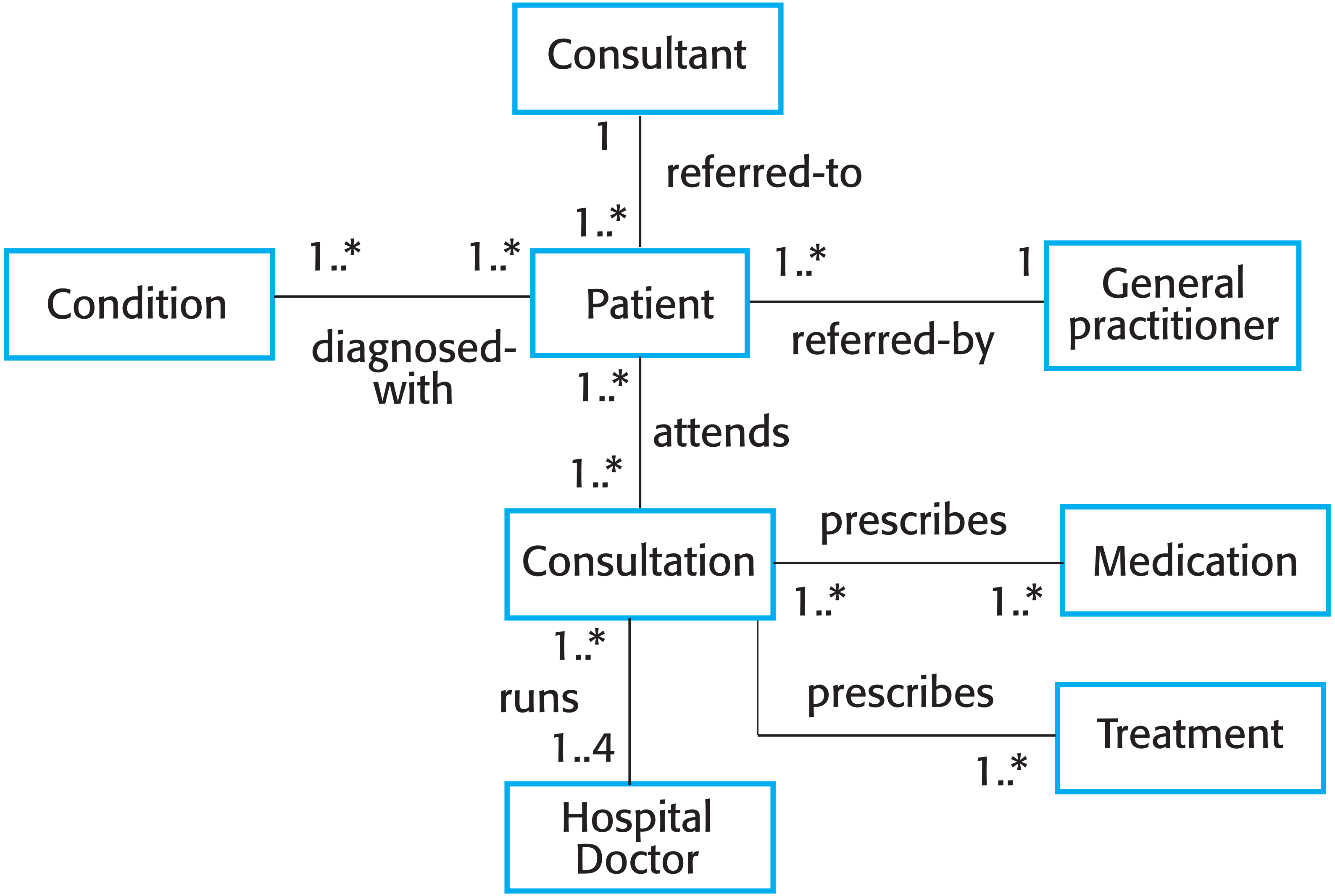
Multiplicity of an Association

- Indicates the number of objects that can participate in a relationship at any point in time



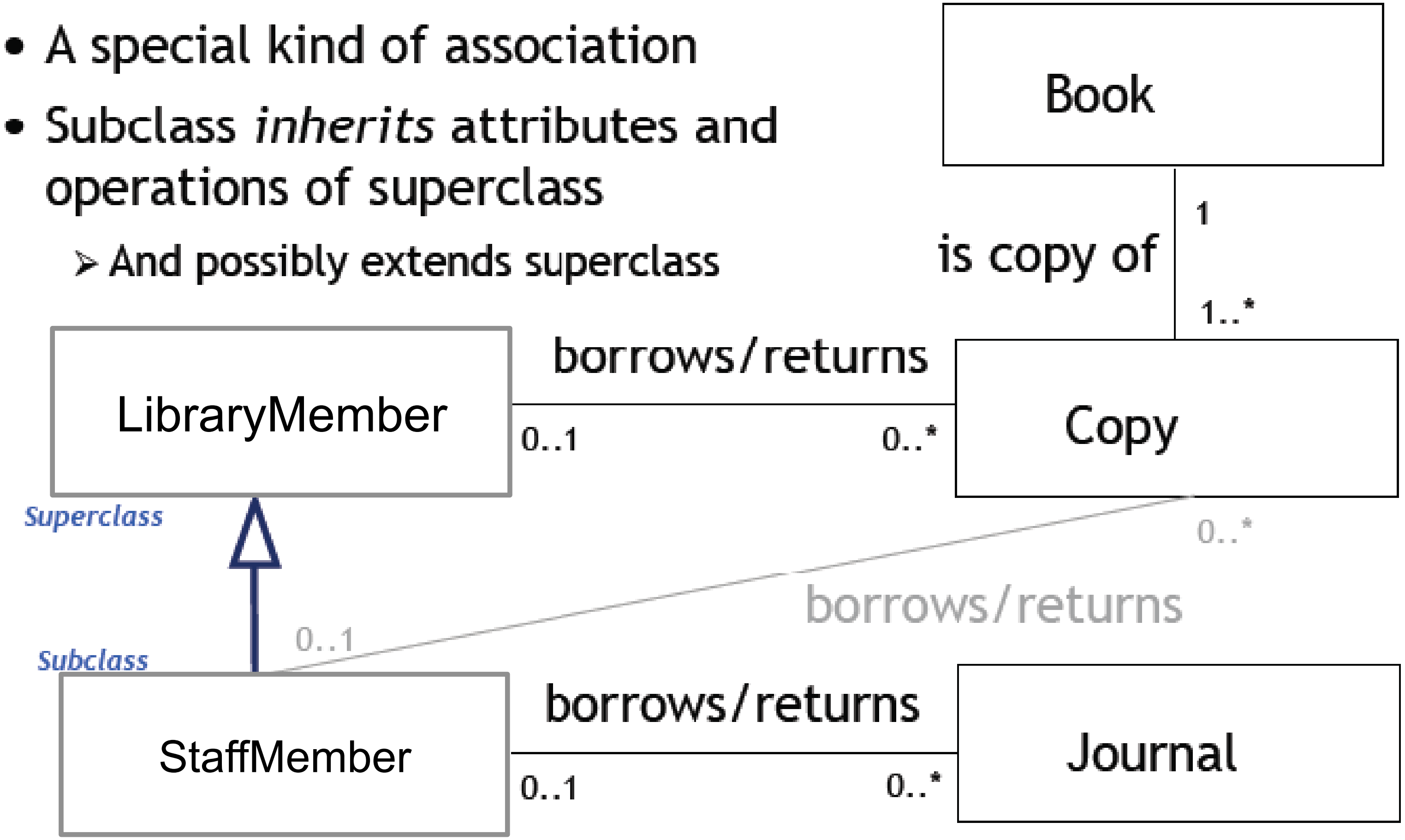
Reference: D. Rosenblum, UCL

Class diagram/Model of the MHC-PMS

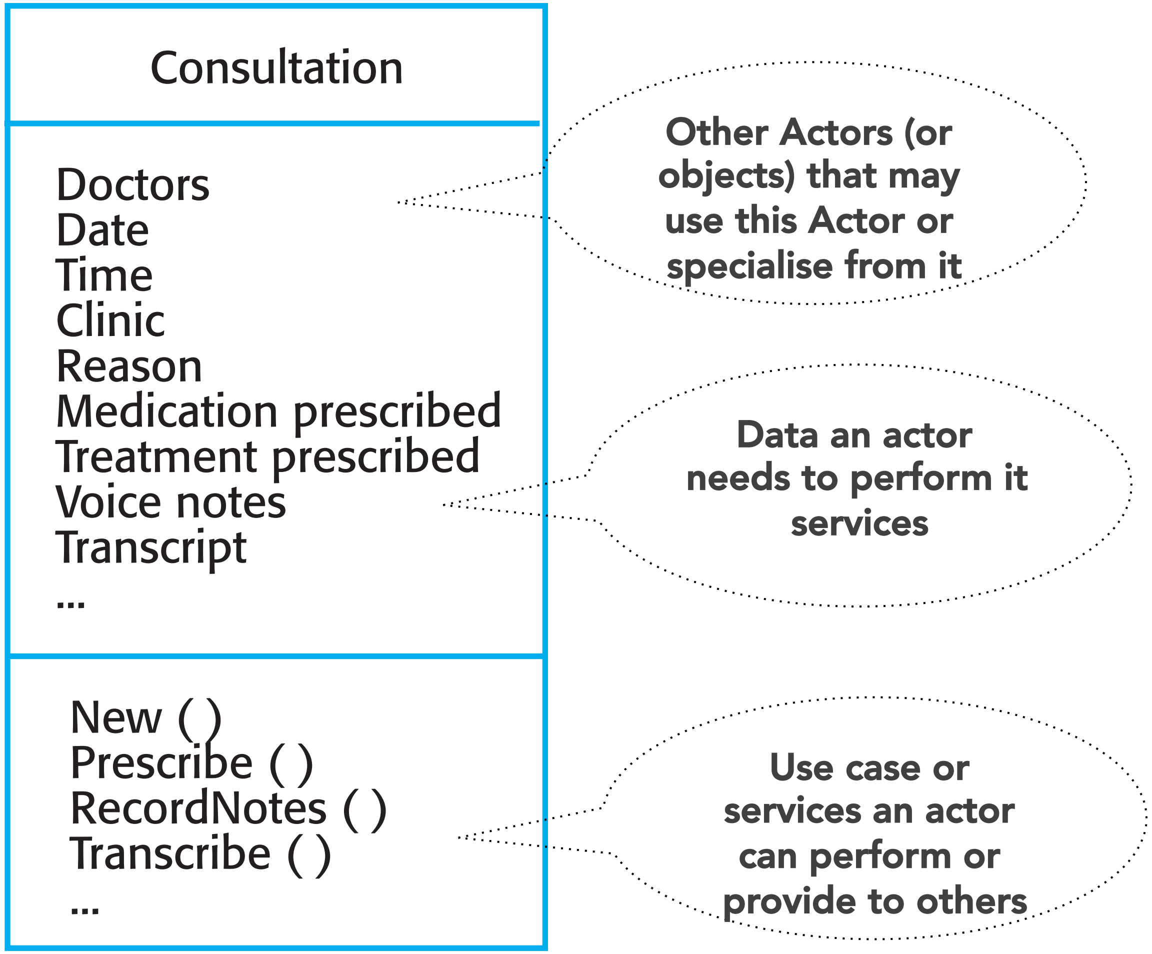


Generalisation (Inheritance)

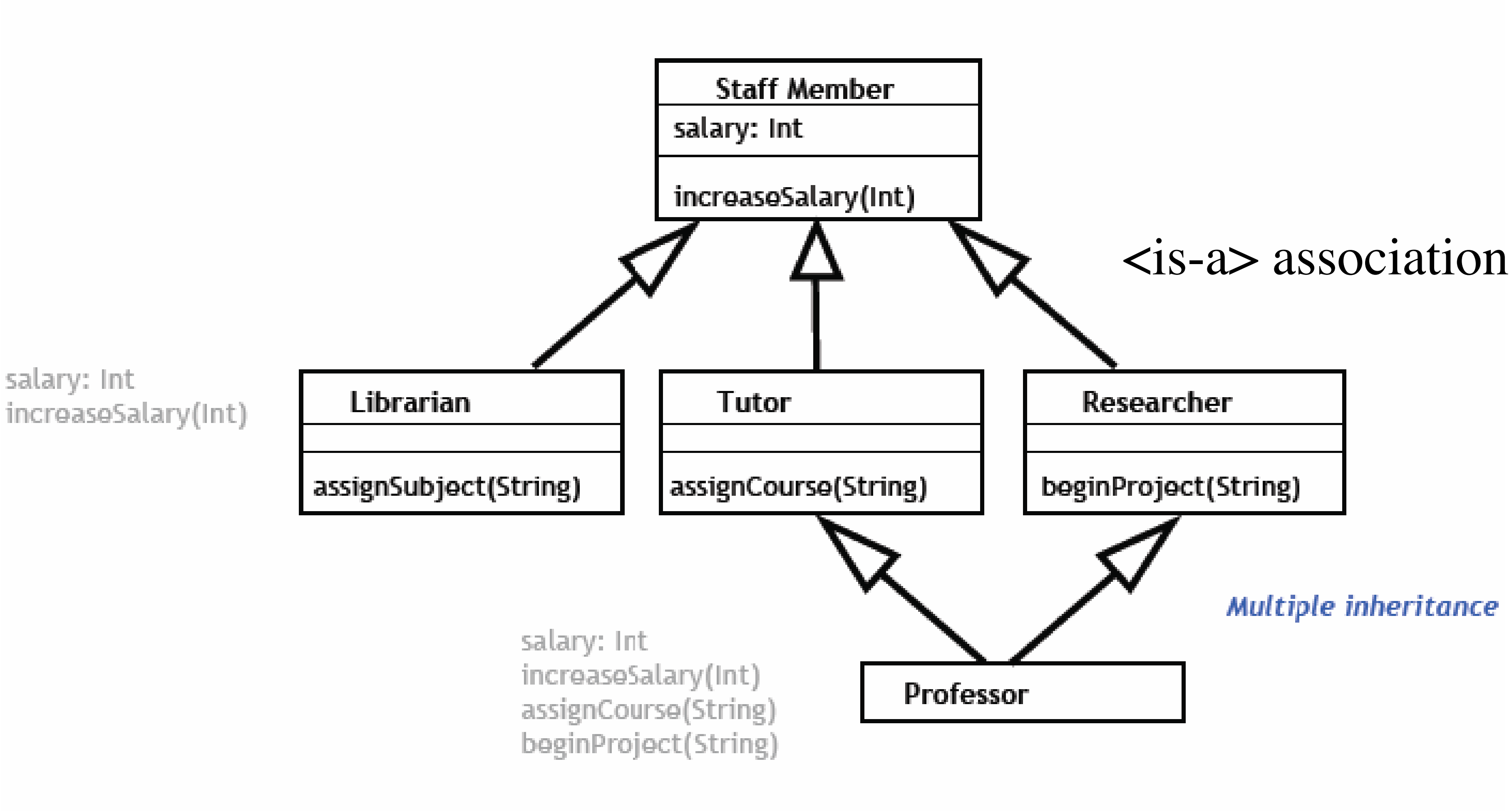
- A special kind of association
- Subclass *inherits* attributes and operations of superclass
  - ➢ And possibly extends superclass



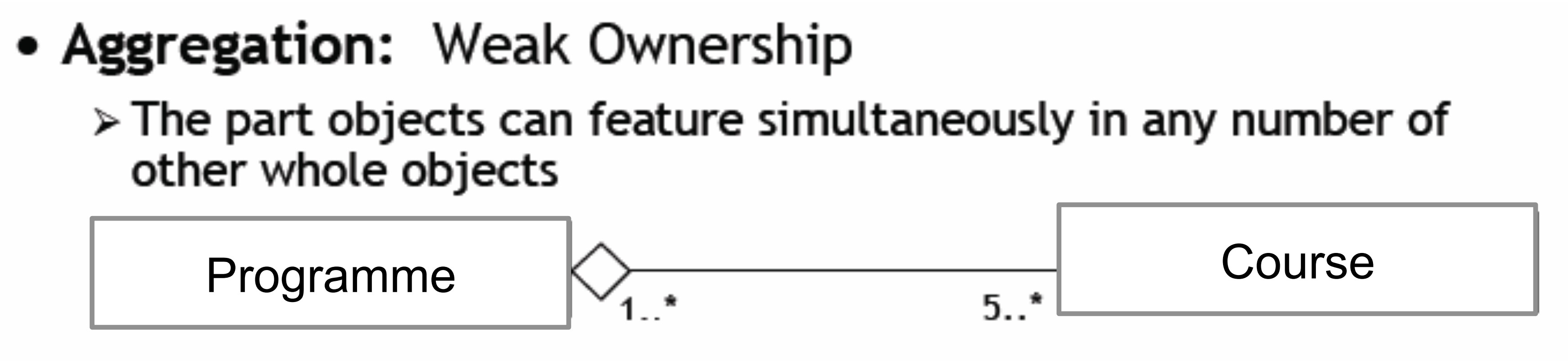
Complete class Description


Consultation
Doctors
Date
Time
Clinic
Reason
Medication prescribed
Treatment prescribed
Voice notes
Transcript
...
New ( )
Prescribe ( )
RecordNotes ( )
Transcribe ( )
...
Other Actors (or objects) that may use this Actor or specialise from it
Data an actor needs to perform it services
Use case or services an actor can perform or provide to others

Another Generalisation Example



Part/Whole Associations (Aggregation)

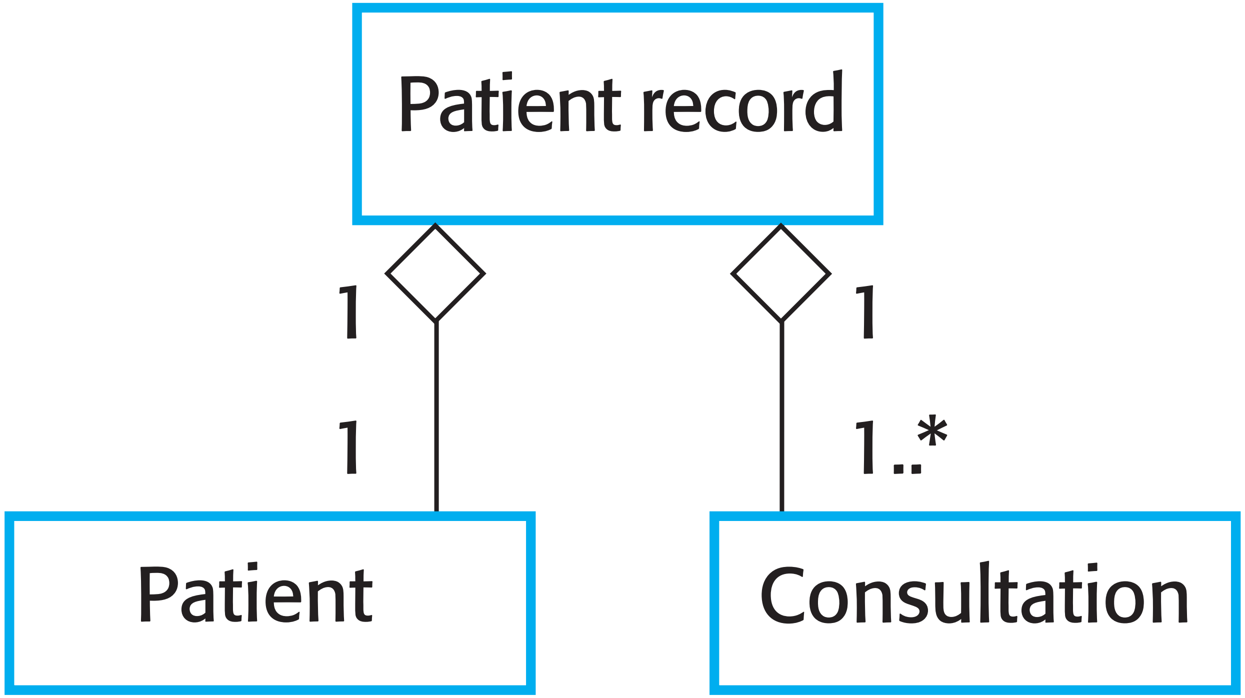


- **Aggregation:** Weak Ownership
  - ➢ The part objects can feature simultaneously in any number of other whole objects

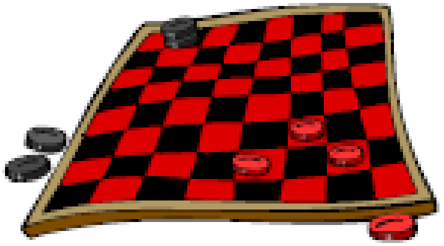
<made-up-of> association
<consist-of> association

a Course is part of a Programme
In fact,
5 or more courses are part of one or more programmes

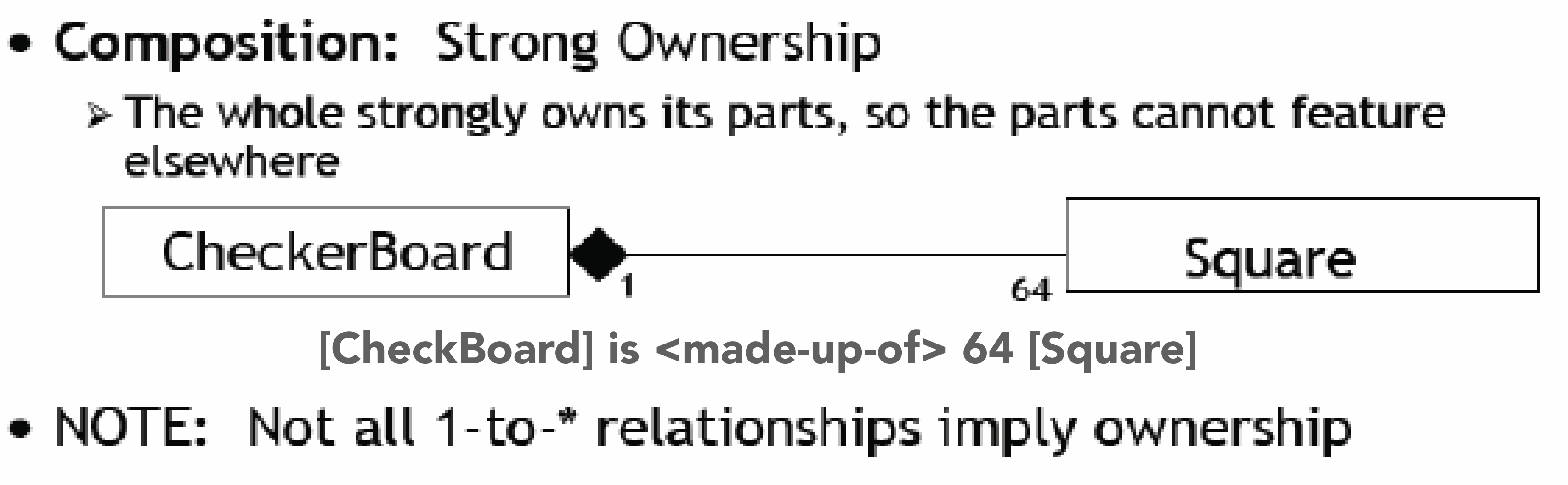
aggregation association: Example



Part/Whole Associations: Example



Composed of 64 squares



- **Composition:** Strong Ownership
  - The whole strongly owns its parts, so the parts cannot feature elsewhere

CheckerBoard 1 — 64 Square

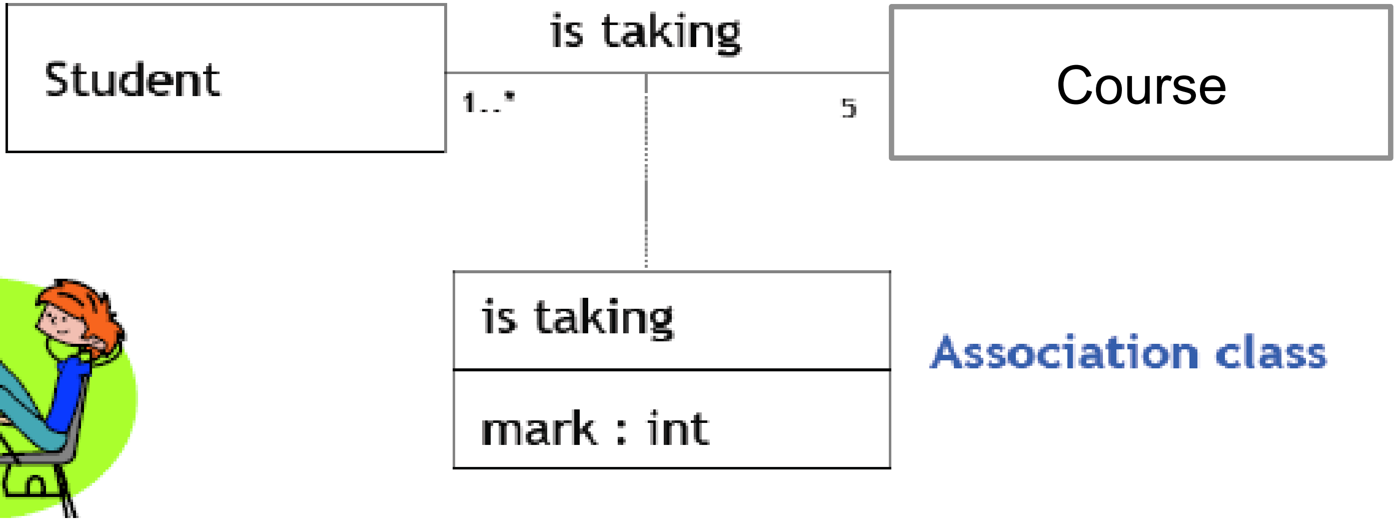
[CheckBoard] is <made-up-of> 64 [Square]

- NOTE: Not all 1-to-* relationships imply ownership

Association Classes

Used to attach attributes to an association itself rather than the classes themselves

Class association line must have the same name!



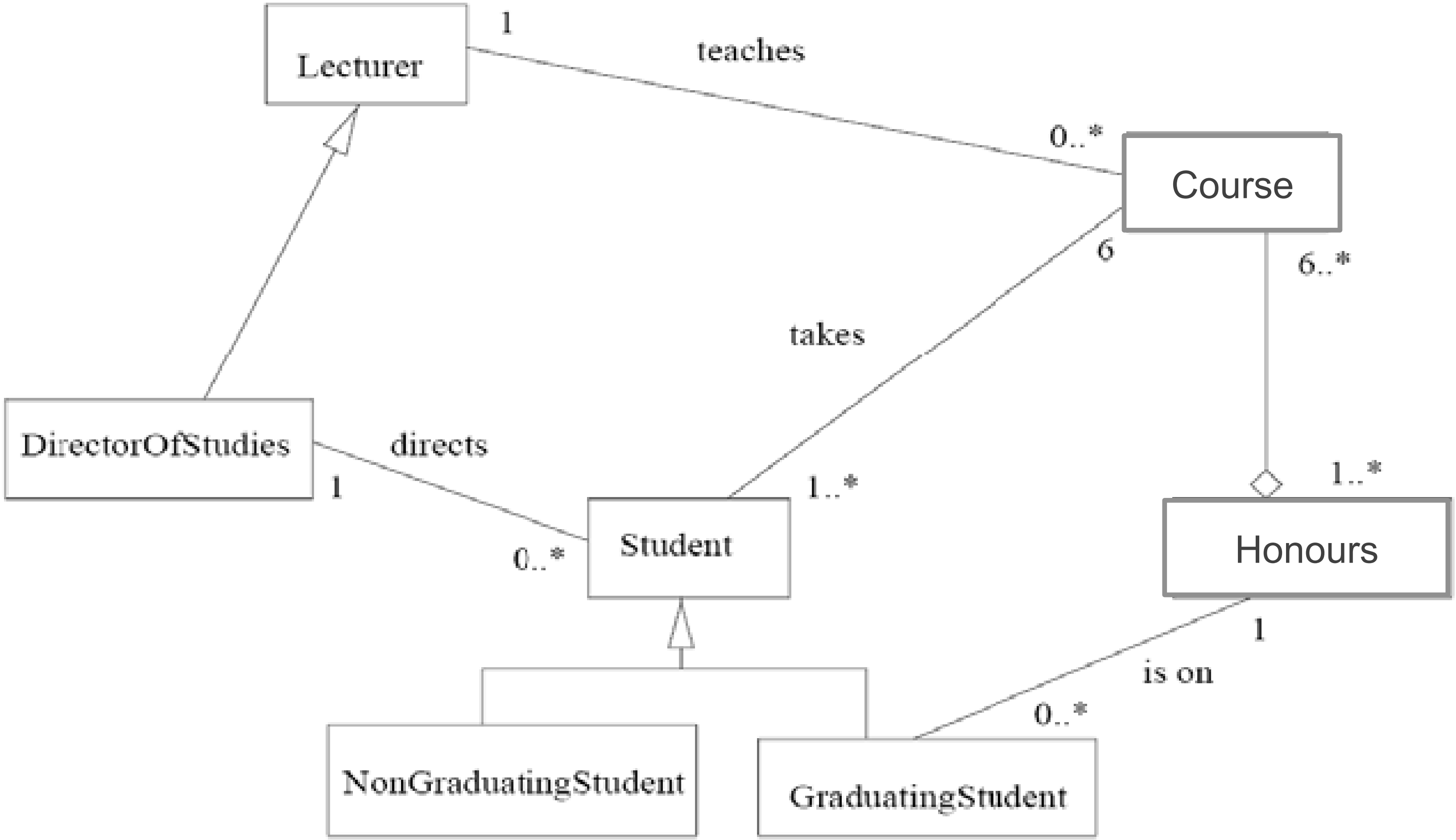
Exercise: Class Model

Students take courses as part of their degree. Some lecturers can teach as many courses as they wish, other can choose not to teach any course. Director of studies is one of the lecturers, who directs students' studies and help them in their course selection. Students can be graduates or non-graduates. Graduate student can graduate with an honours degree, or a non-honour degree for their graduation year. Students with honours should pass at least 6 courses, in their final graduating year in their speciality, with a mark of "very good (or first class)" and above to gain an honour degree.

Exercise: Class Model

Students take courses as part of their degree. Some lecturers can teach as many courses as they wish, other can choose not to teach any course. Director of studies is one of the lecturers, who directs students' studies and help them in their course selection. Students can be graduates or non-graduates. Graduate student can graduate with an honours degree, or a non-honour degree for their graduation year. Students with honours should pass at least 6 courses, in their final graduating year in their speciality, with a mark of "very good (or first class)" and above to gain an honour degree.

Example: (Analysis) Class Model



What Makes a 'Good' Analysis Class..

Its name reflects its intent

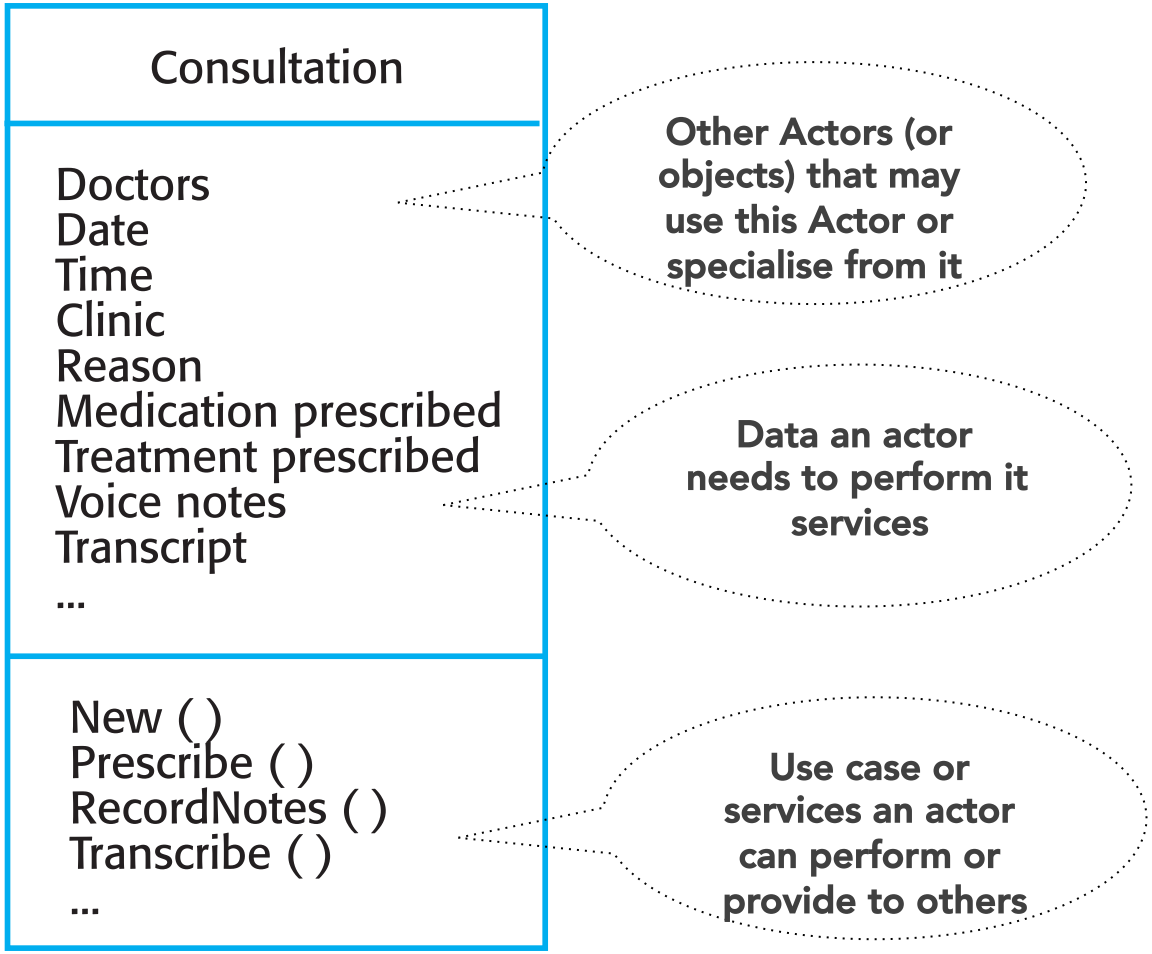
It is a crisp abstraction that models one specific element of the problem domain

It has a small but defined set of responsibilities

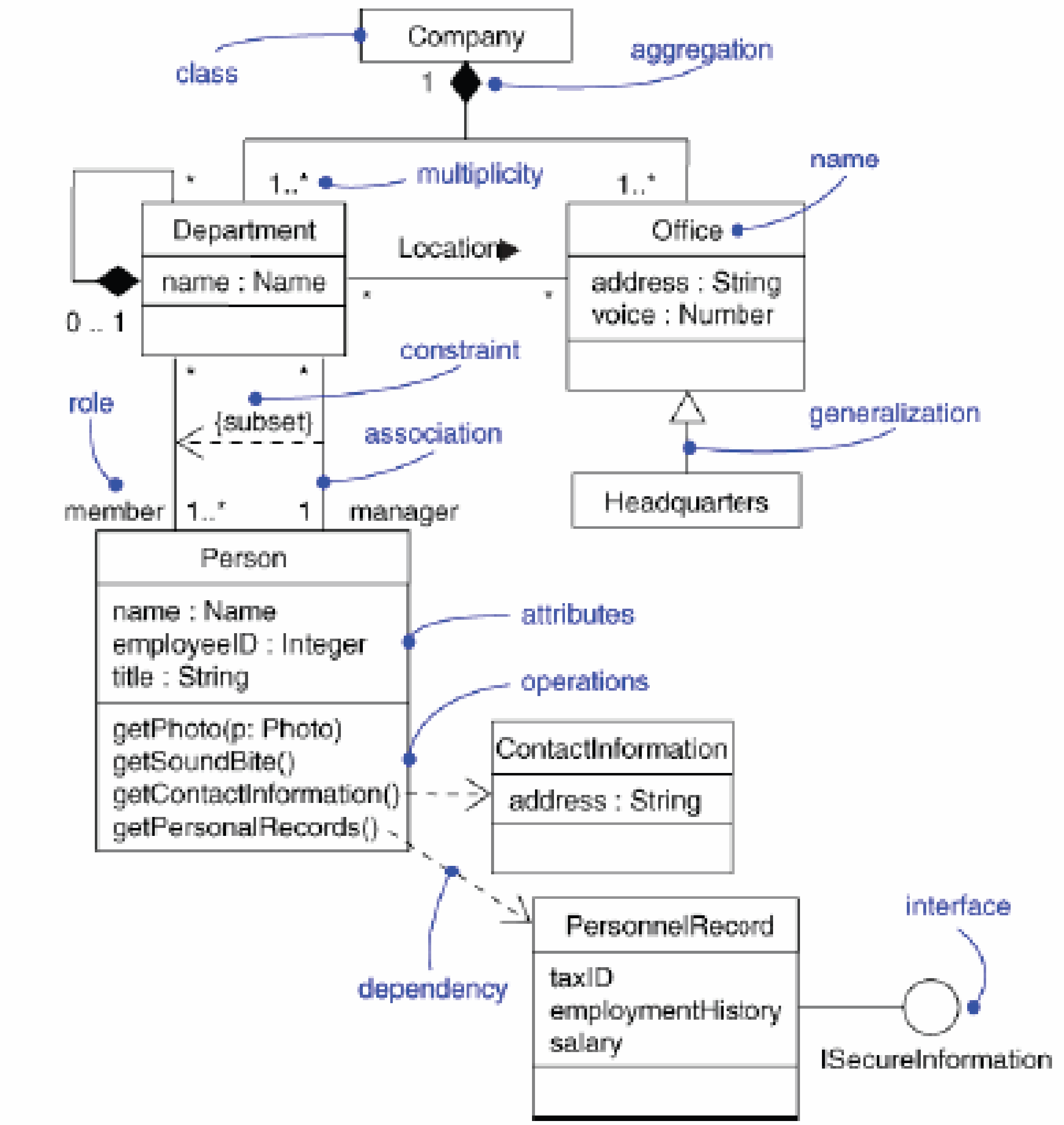
It has high *cohesion*

It has low *coupling* with other classes

Complete class Description

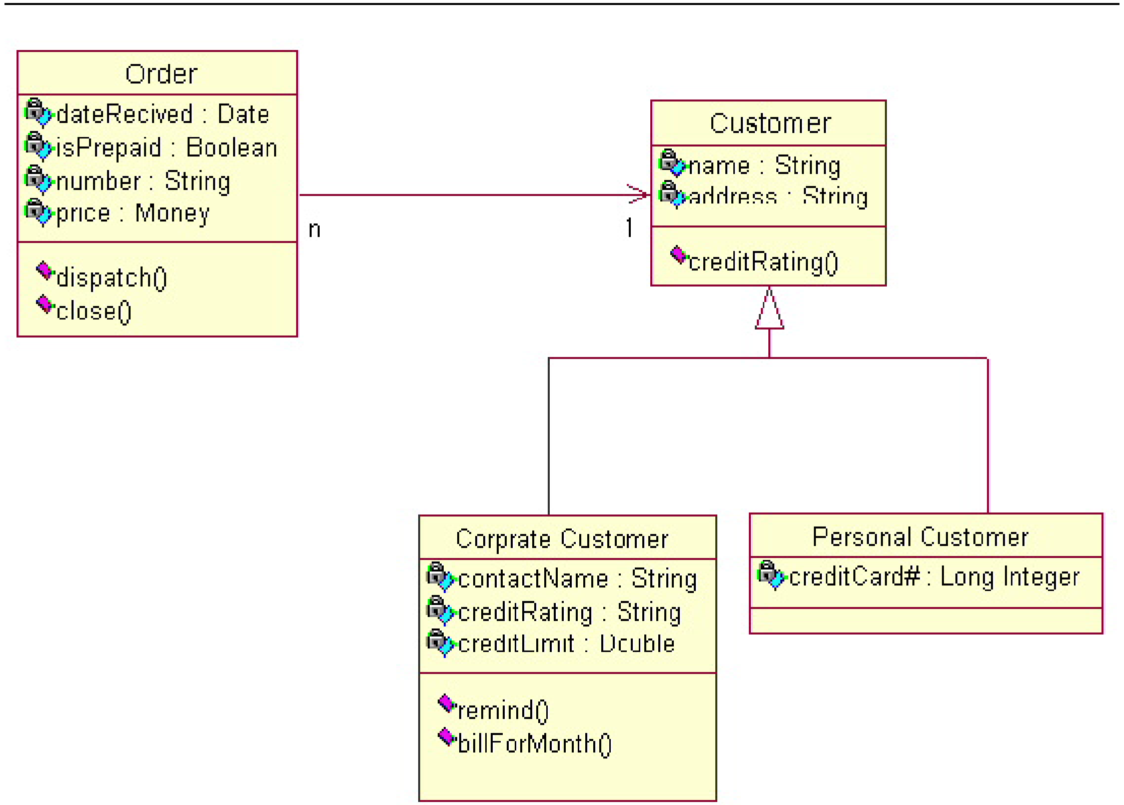

Consultation
Doctors
Date
Time
Clinic
Reason
Medication prescribed
Treatment prescribed
Voice notes
Transcript
...
New ( )
Prescribe ( )
RecordNotes ( )
Transcribe ( )
...
Other Actors (or objects) that may use this Actor or specialise from it
Data an actor needs to perform it services
Use case or services an actor can perform or provide to others

Example: Detailed Class Diagram

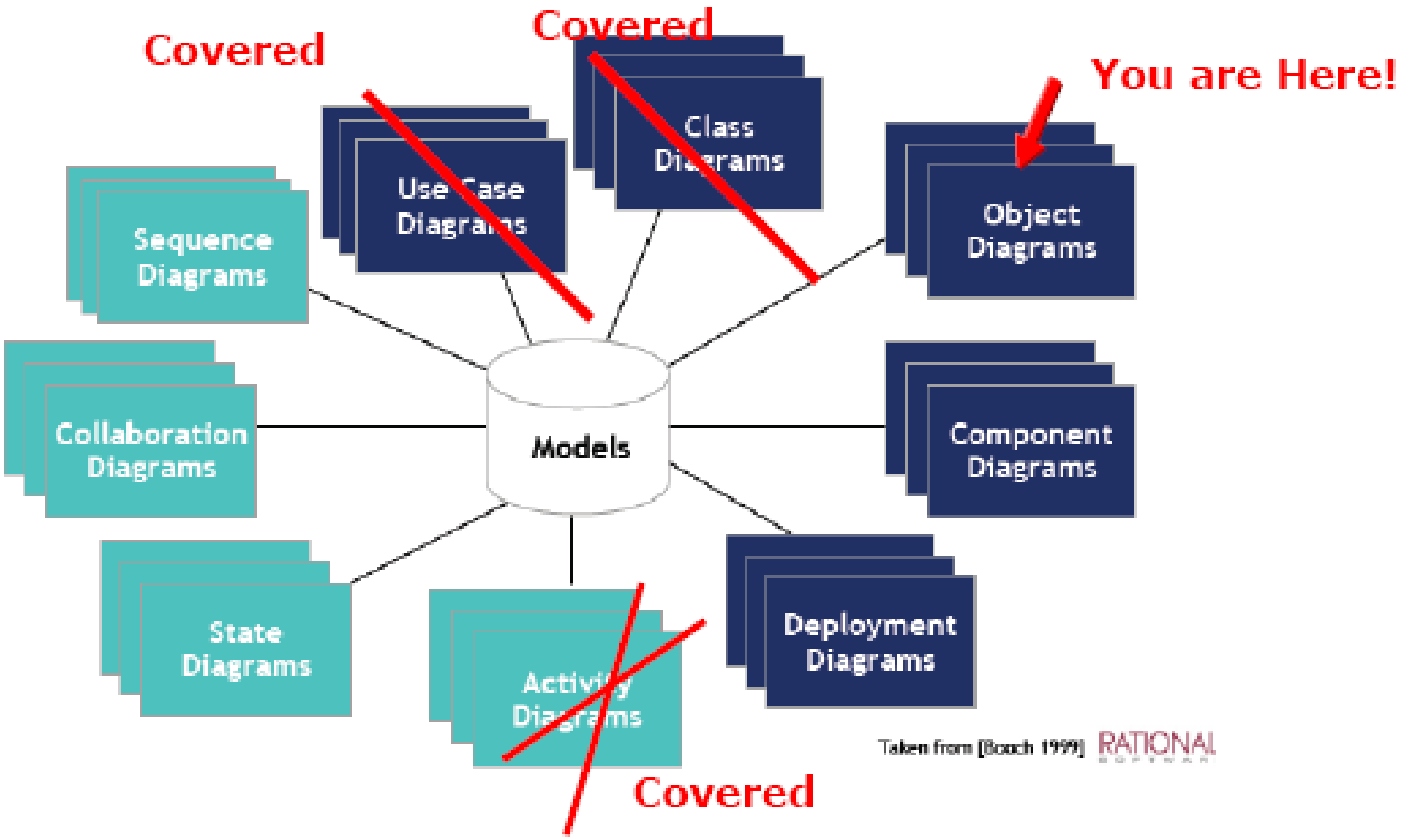


Another Example

Corporate Customer and Personal Customer classes may have some common attributes/operations such as name and address, but each class has its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes.



UML Diagrams



Taken from [Booch 1999]

Object Diagram

Objects are instances of Classes

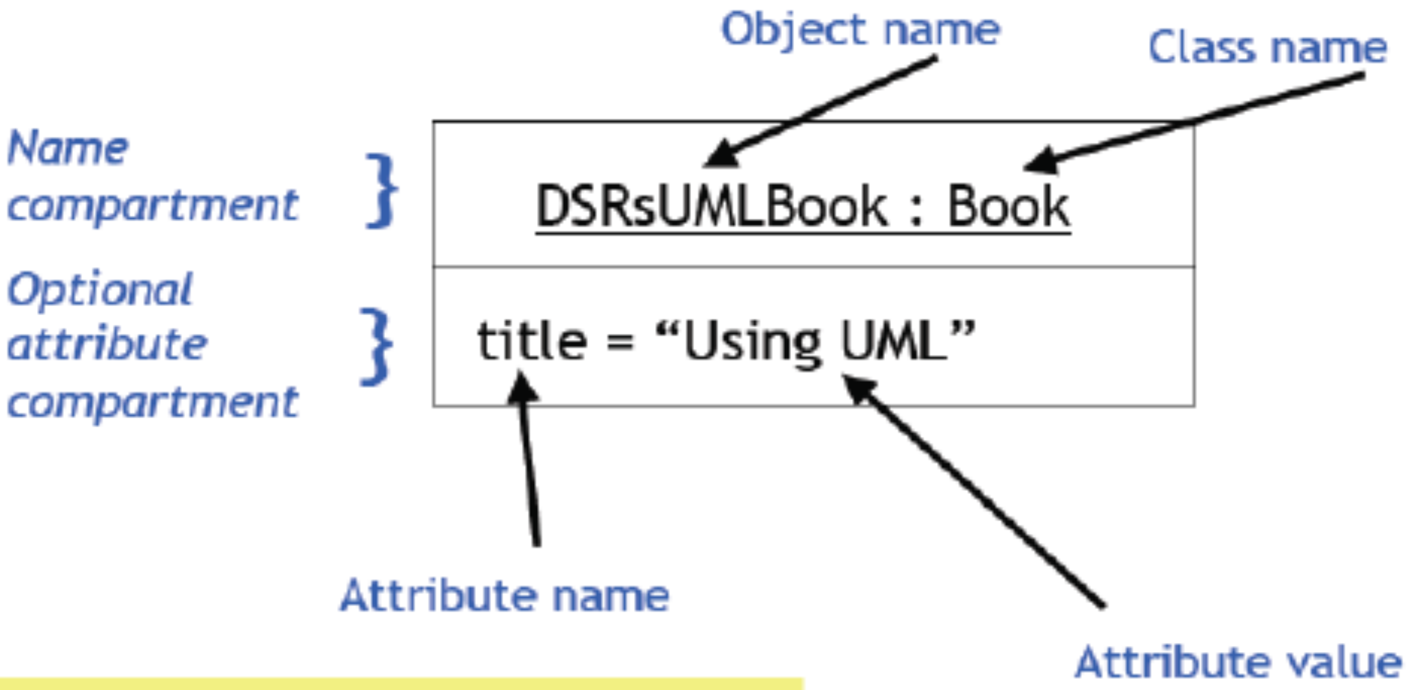
Object Diagram captures objects and relationships between them, in other words, it captures instances of Classes and links between them.

Built during analysis & design

- Illustrate data/object structures
- Specify snapshots

Developed by analysts, designers and implementers

UML Object Icons

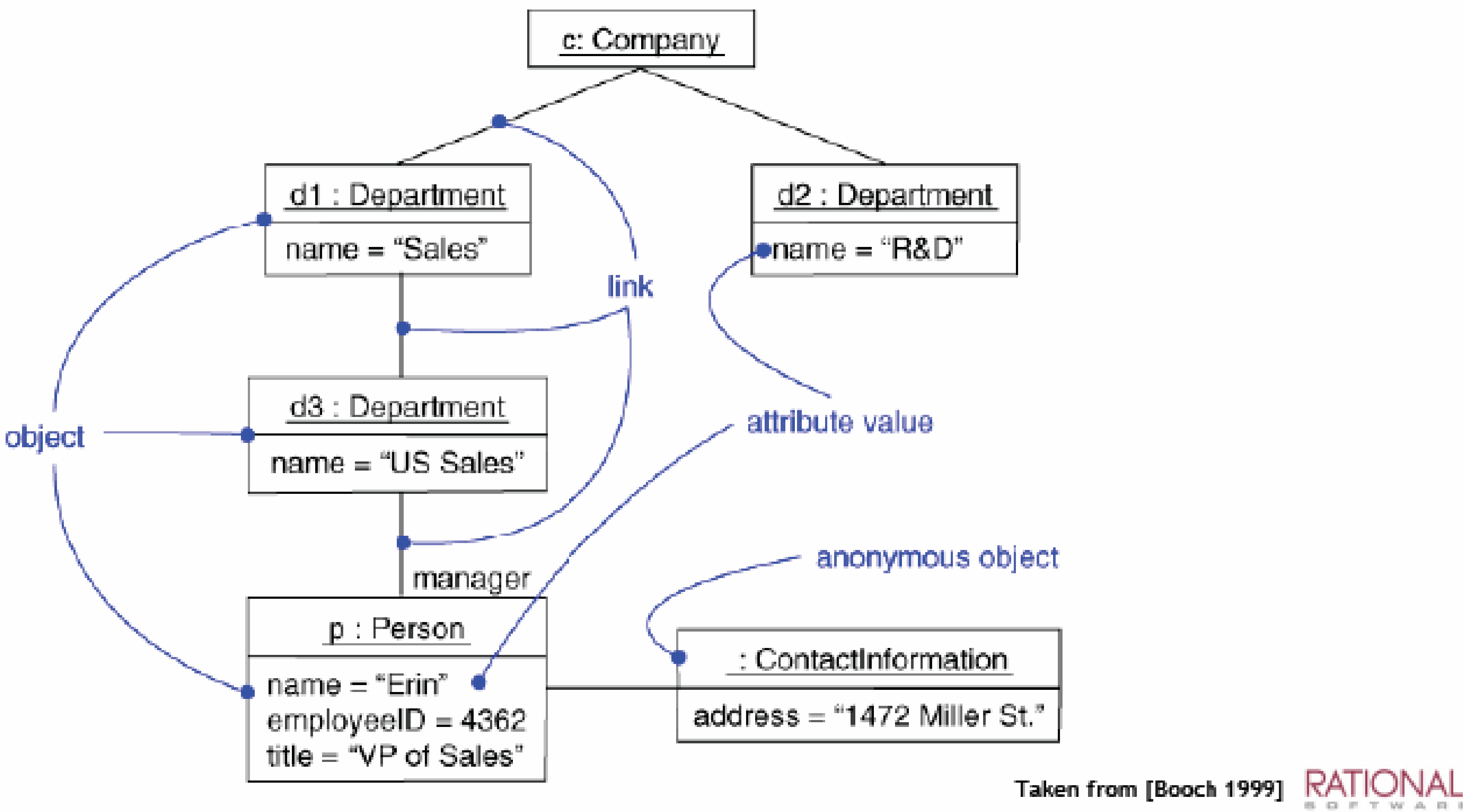


Operations and attribute types are *not* shown on object diagrams!

Reference: D. Rosenblum, UCL

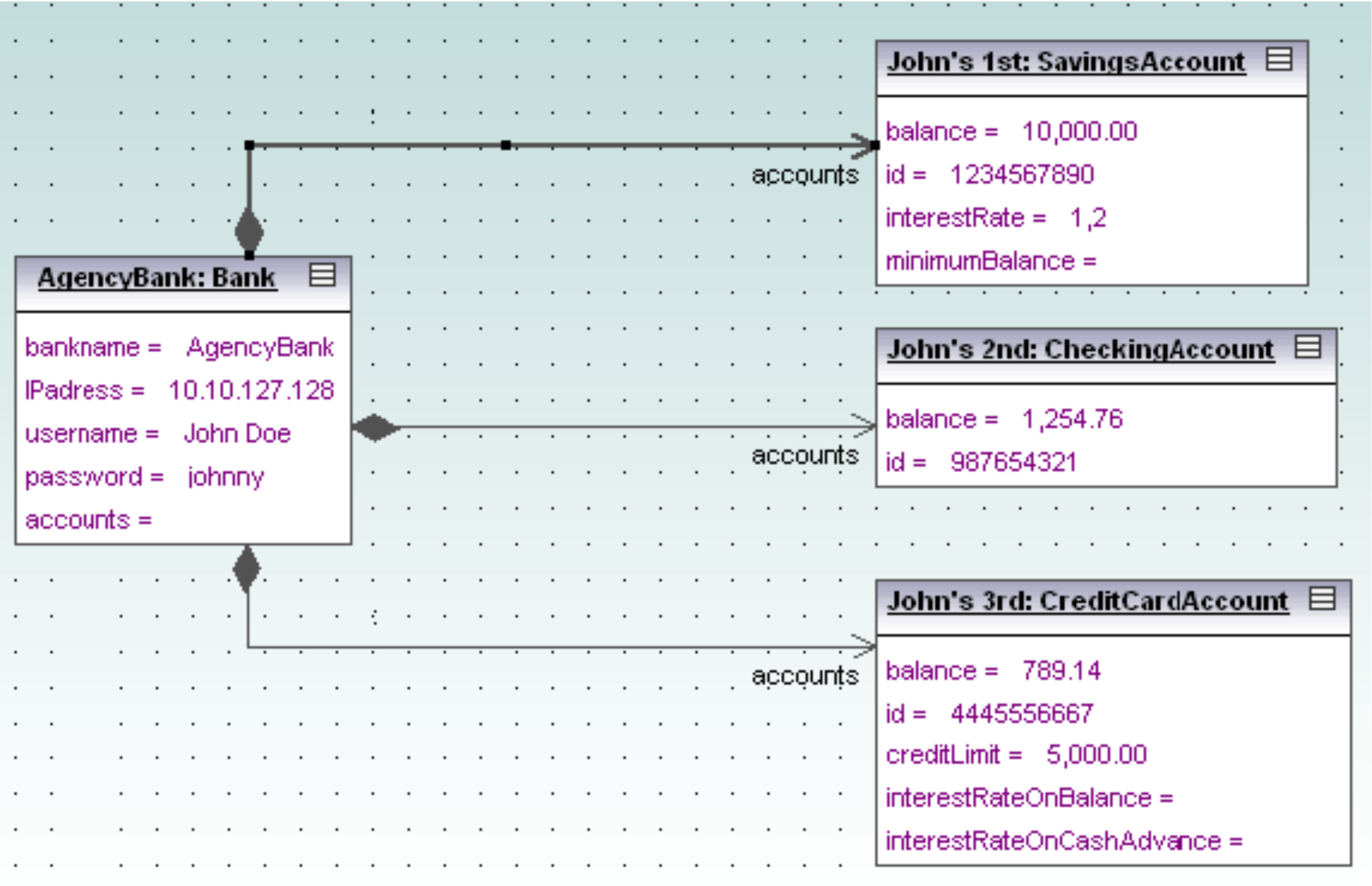
Object Diagram

Capture *class instances* and *links* between objects



Taken from [Booch 1999]

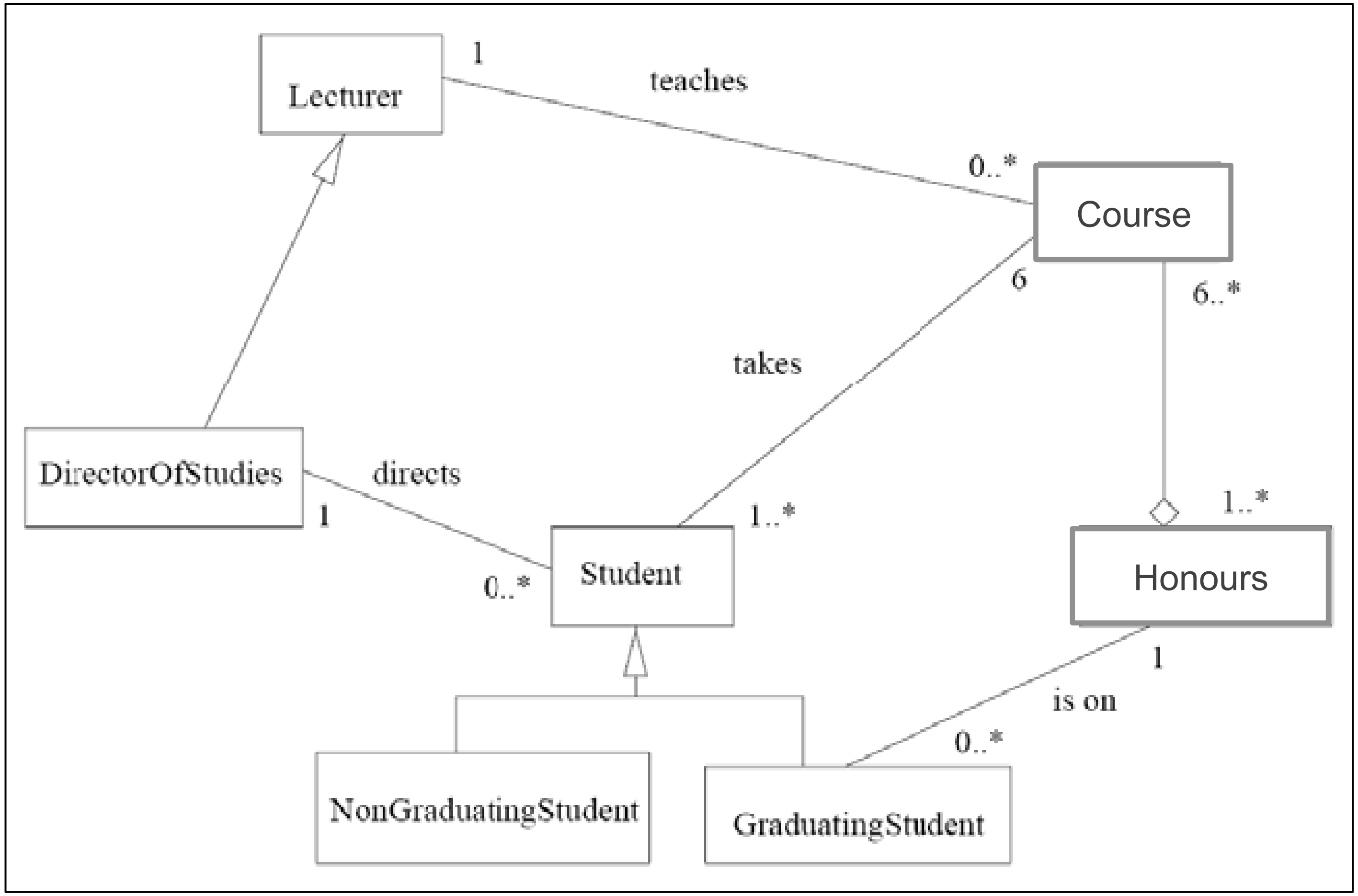
Example: Object Diagram



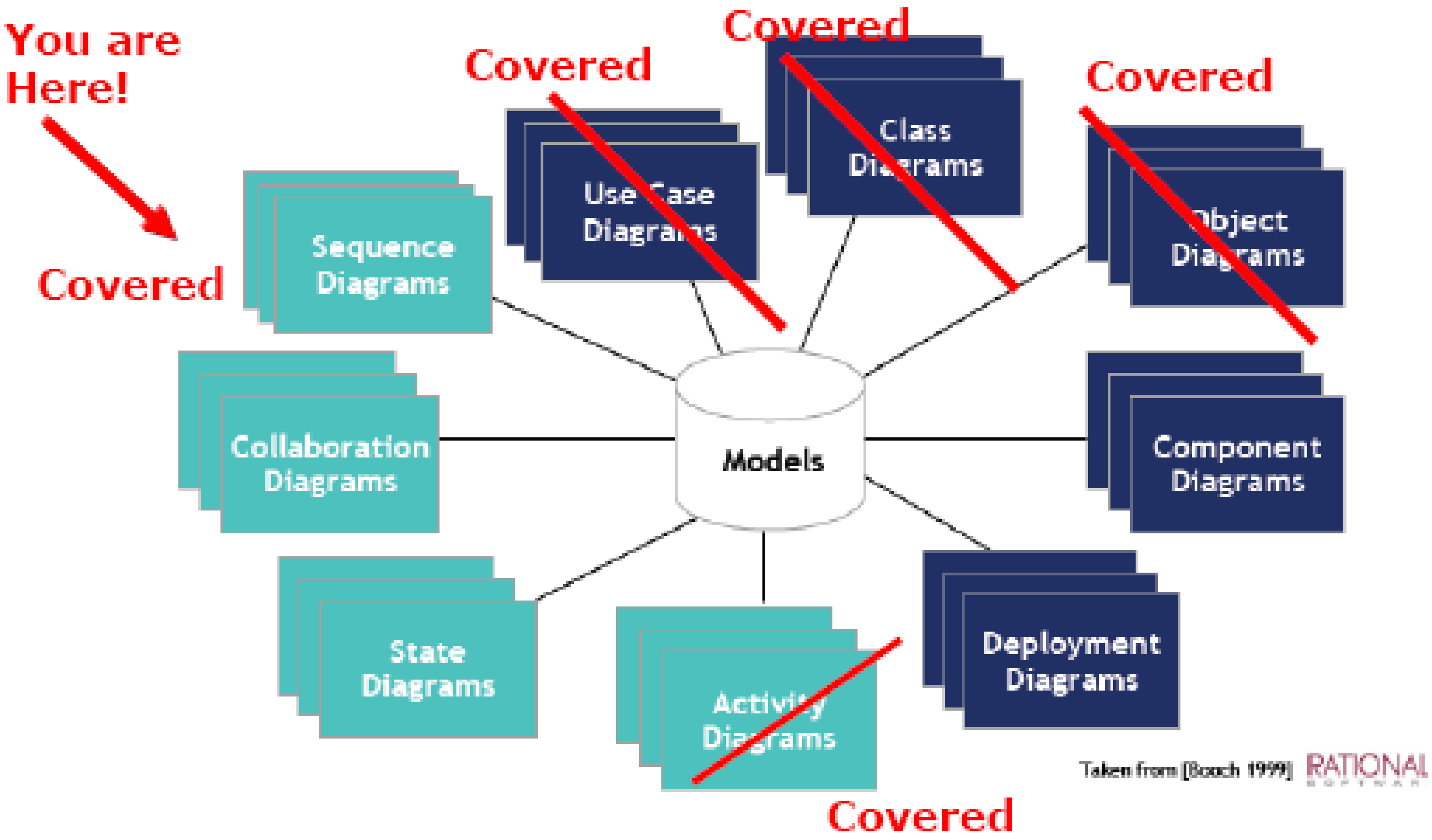
Example: Object Model/Diagram

For the following class model draw:

- a detailed Class Model (or Diagram)
- an Object Model (or Diagram)



UML Diagrams



Taken from [Booch 1999]

Sequence diagrams

Sequence diagrams are used to model the interactions between the actors and the objects within a system, with a time-oriented view.

A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

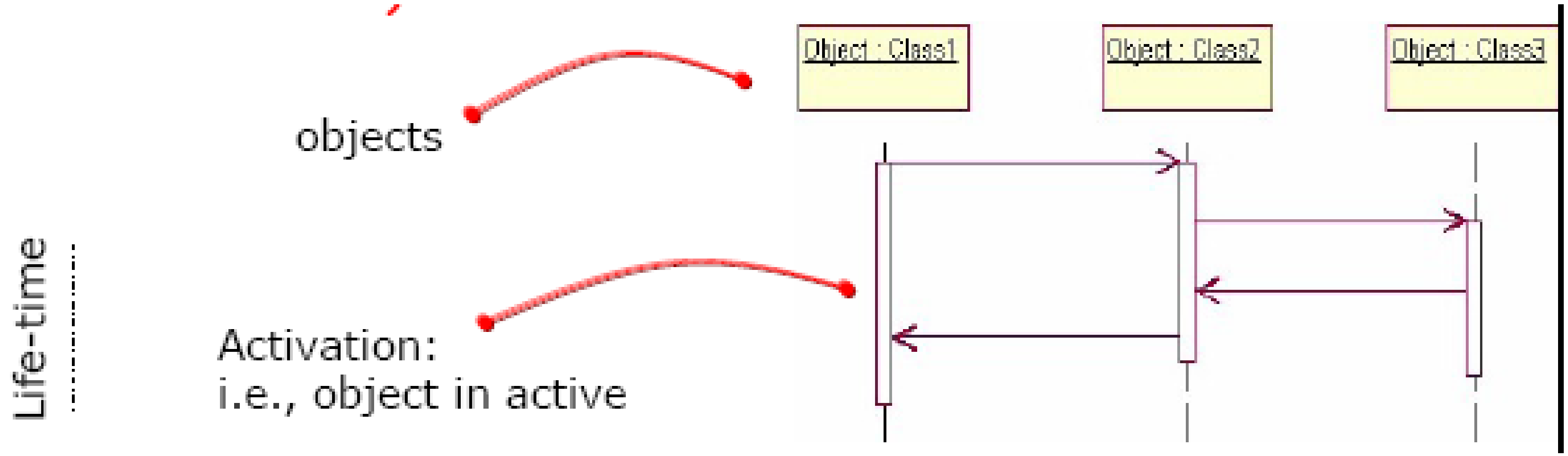
The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.

Interactions between objects are indicated by annotated arrows.

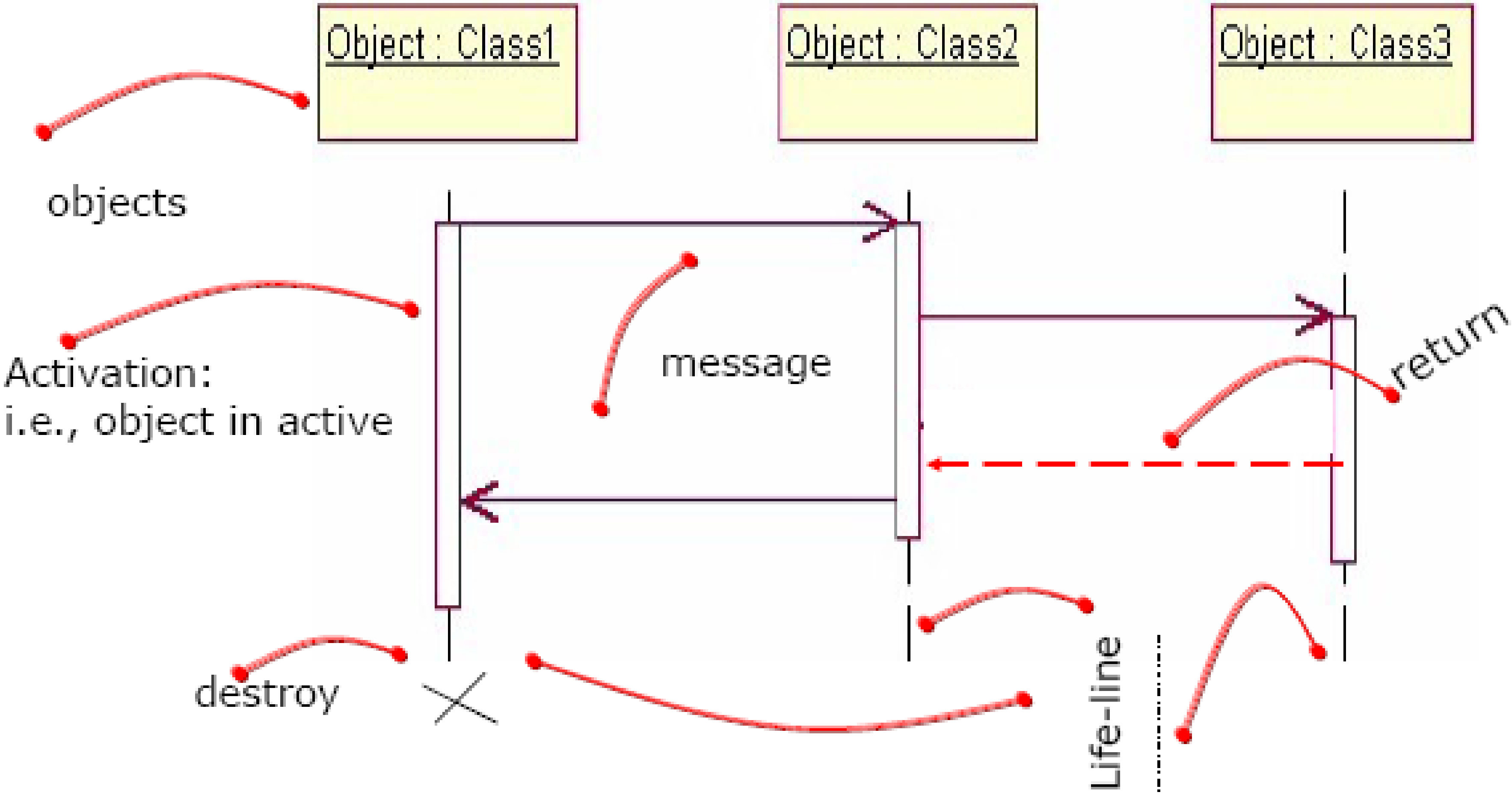
Sequence diagrams

Sequence diagrams demonstrate the behaviour of objects in a use case by describing the objects and the messages they pass. the diagrams are read left to right and descending.

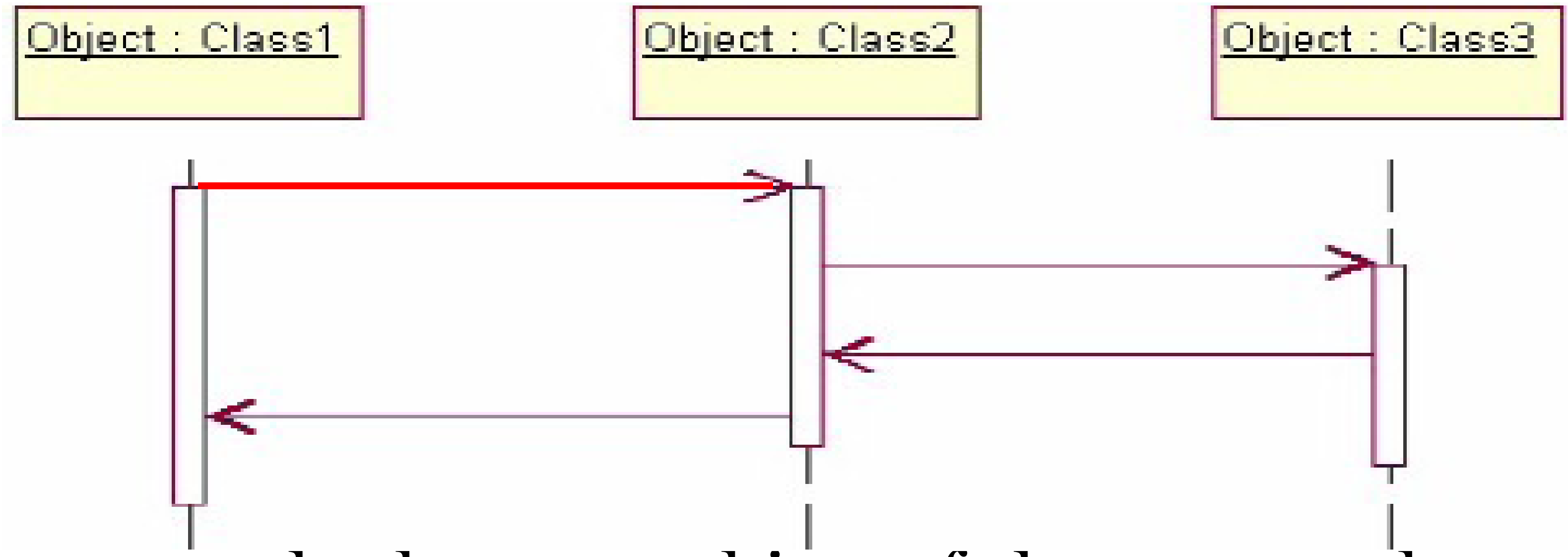
Object interactions are arranged in a time sequence (i.e. time-oriented)



Sequence diagrams



Sequence diagrams



The example shows an object of class 1 start the behaviour by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message

Example

In a self-service, e.g. money (e.g. ATM), machine, three objects do the work we're concerned with:

the front: the interface the self-service machine presents to the customer

the money register: part of the machine where money is collected

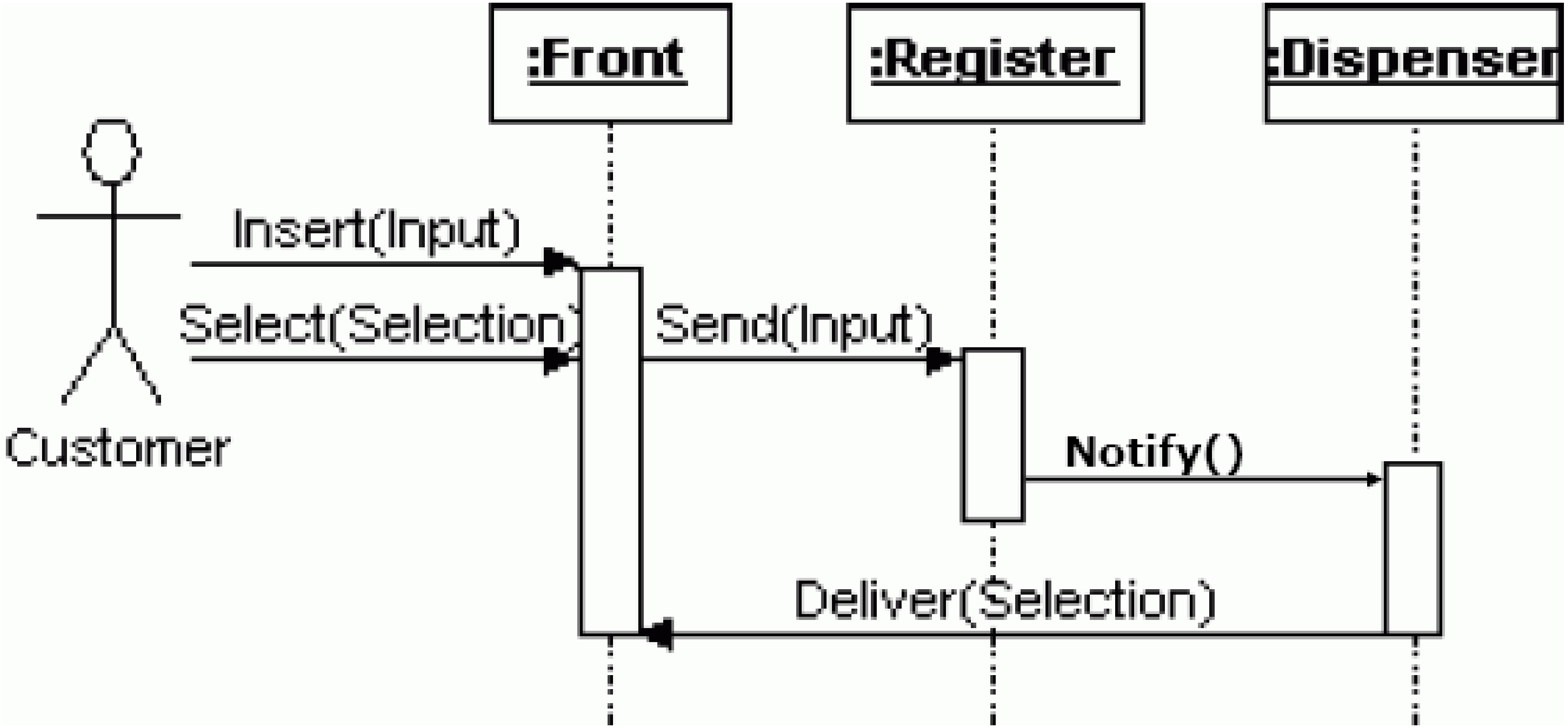
the dispenser: which delivers the selected product to the customer

Example

The instance sequence diagram may be sketched by using this sequences:

1. The customer inserts money in the money slot in **front** money collector.
2. The customer makes a selection on the **front** UI
3. The money travels to the **register**
4. The **register** checks to see whether the correct money is in the money **collector/dispenser**
5. The **register** updates its cash reserve
6. The **register** notifies the **dispenser** which delivers the product (e.g. receipt) to the **front** of the machine

Example



The "Buy a product" scenario.
Because this is the best-case scenario, it's an instance sequence diagram

However, note…

We have seen an instance of an interaction diagram- i.e. one possible sequence of messages

Since a use case can include many scenarios

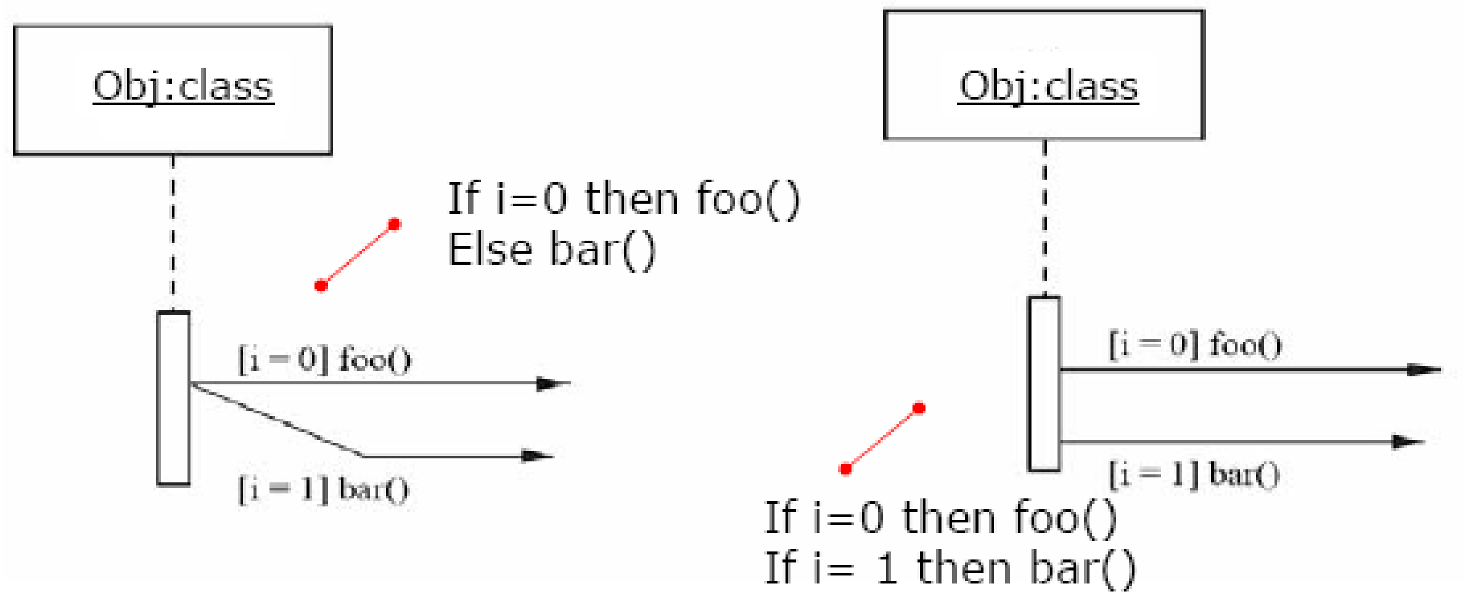
- There is a need to show conditional behaviour
- There is a need to show possible iterations

A generic interaction diagram shows all possible sequences of messages that can occur

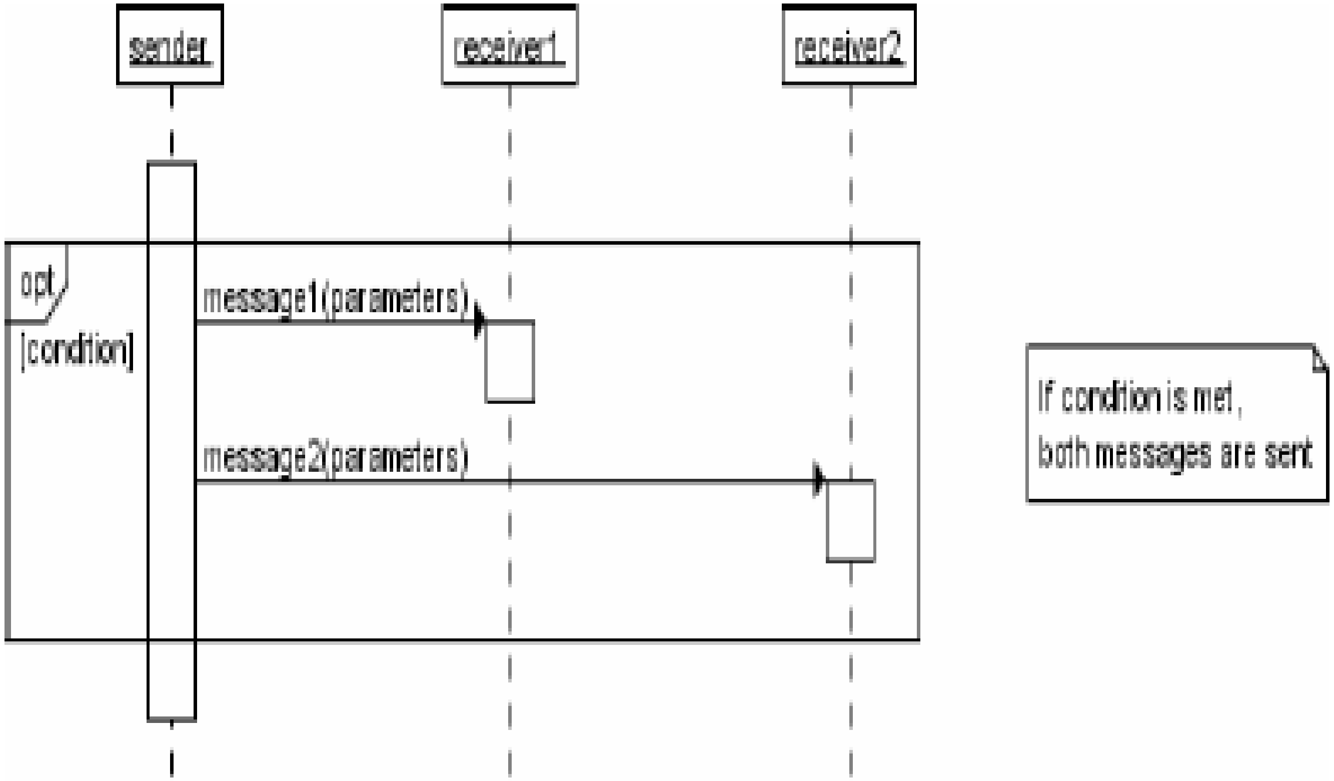
Showing conditional behaviour

A message may be **guarded** by a condition

Messages are only sent if the **guard** evaluates to true at the time when the system reaches that point in the interaction

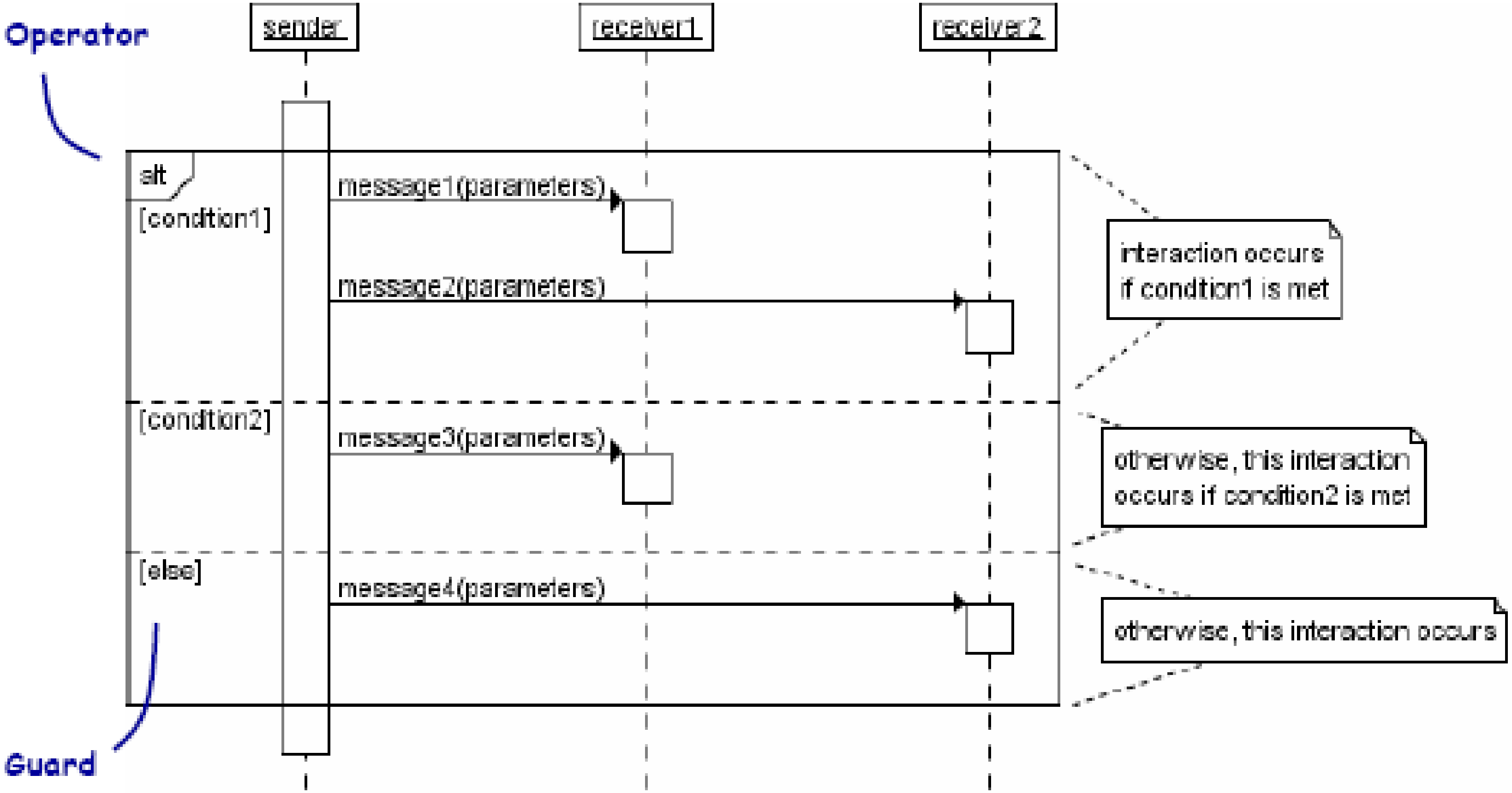


Opt(ional) in UML 2.0



Opt: Optional; the fragment executes only if the supplied condition is true. This is equivalent to an alt with one trace

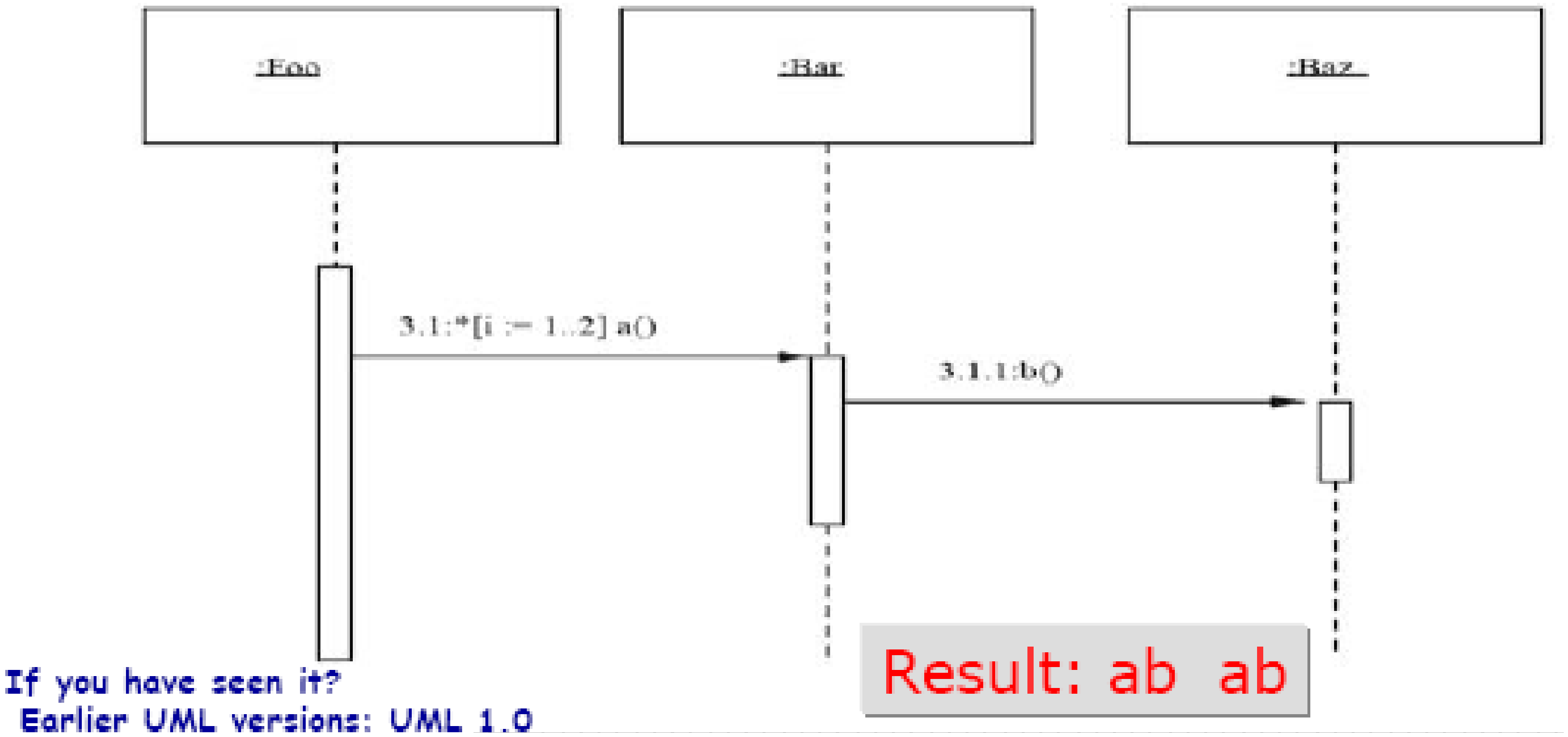
alt(ernative): Operators in interactions frames – UML 2.0



Alternative multiple fragment: only the one whose condition is true will execute

Iterations (i.e., loop) – UML 1.0

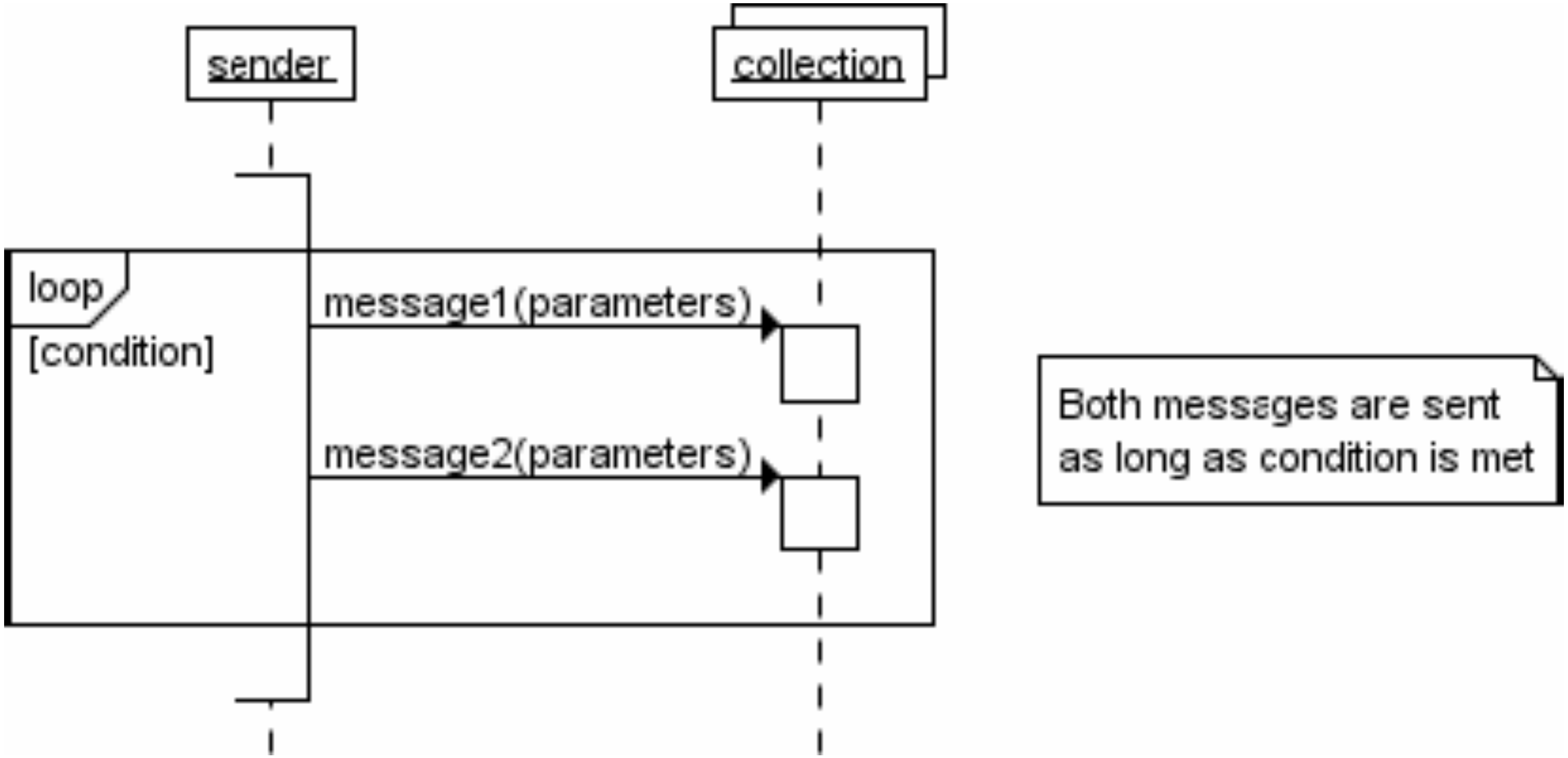
* Indicates looping or iterations
i:=1..2 means 2 iterations….



If you have seen it?
Earlier UML versions: UML 1.0

Result: ab ab

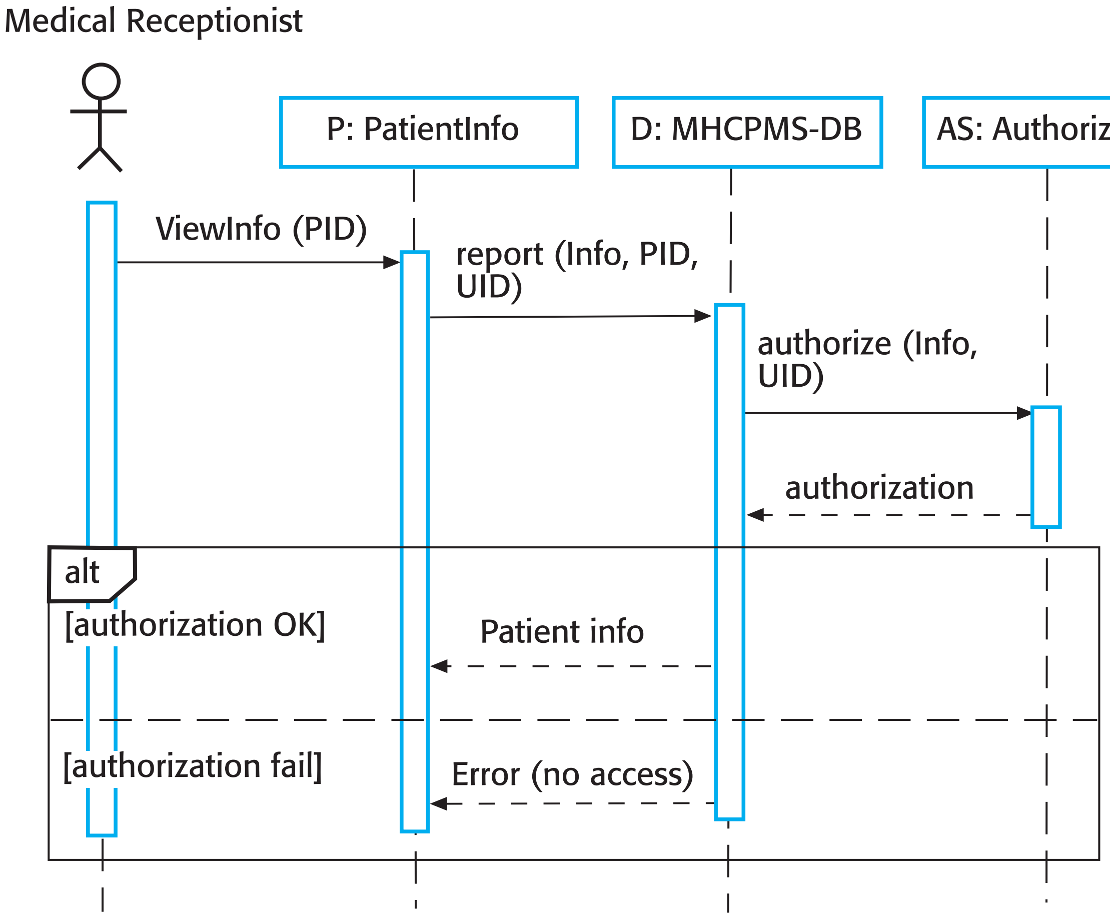
Loop in UML 2.0



Loop: the fragment may execute multiple times, and the guard indicates basis for iterations

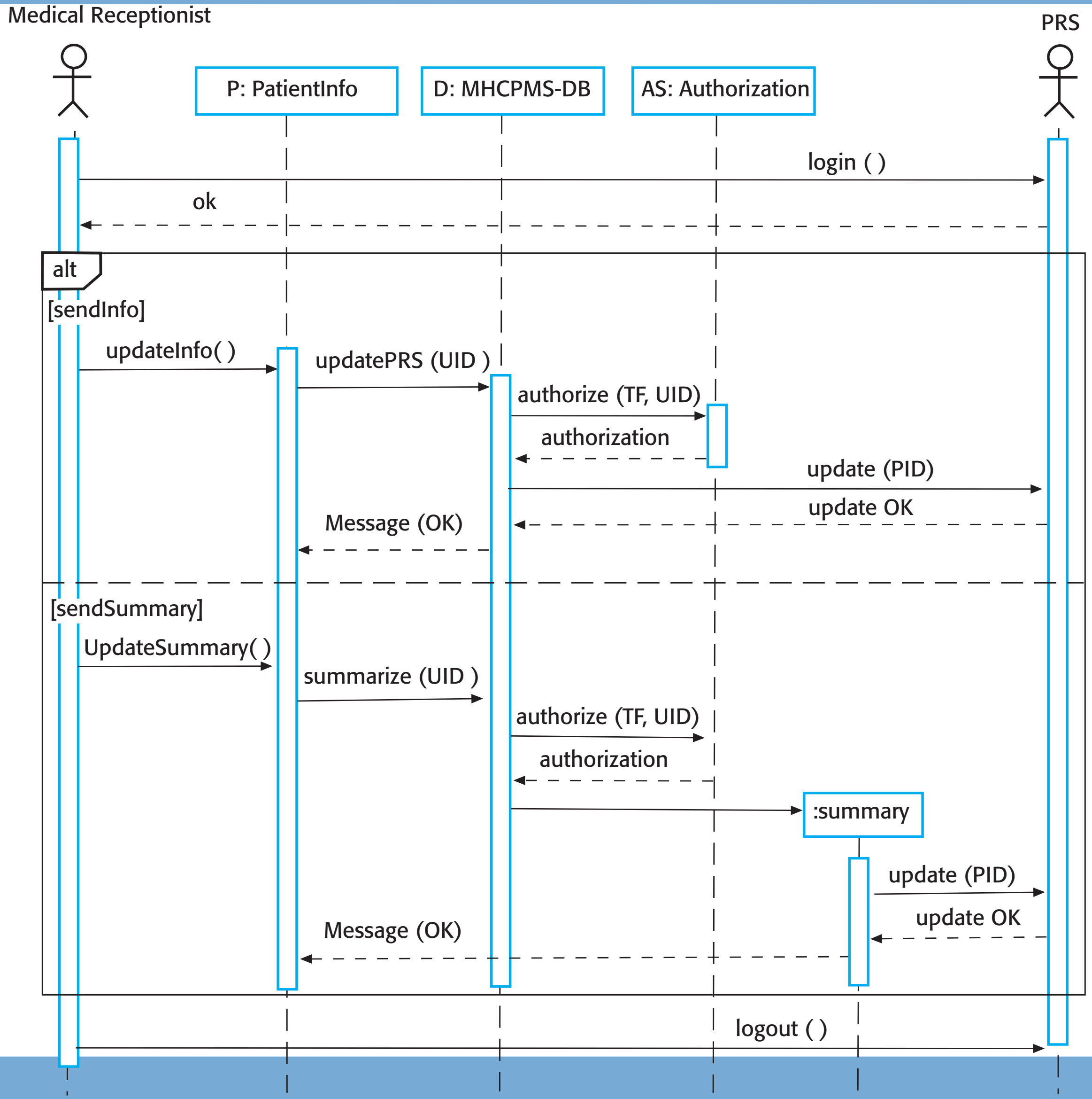
Sequence diagram for View patient information use case

Use case: View Patient Information – through authorization



Sequence diagram for Transfer Data

Use case: Transfer Data- demonstrates interactions between Actors



Example/Exercise

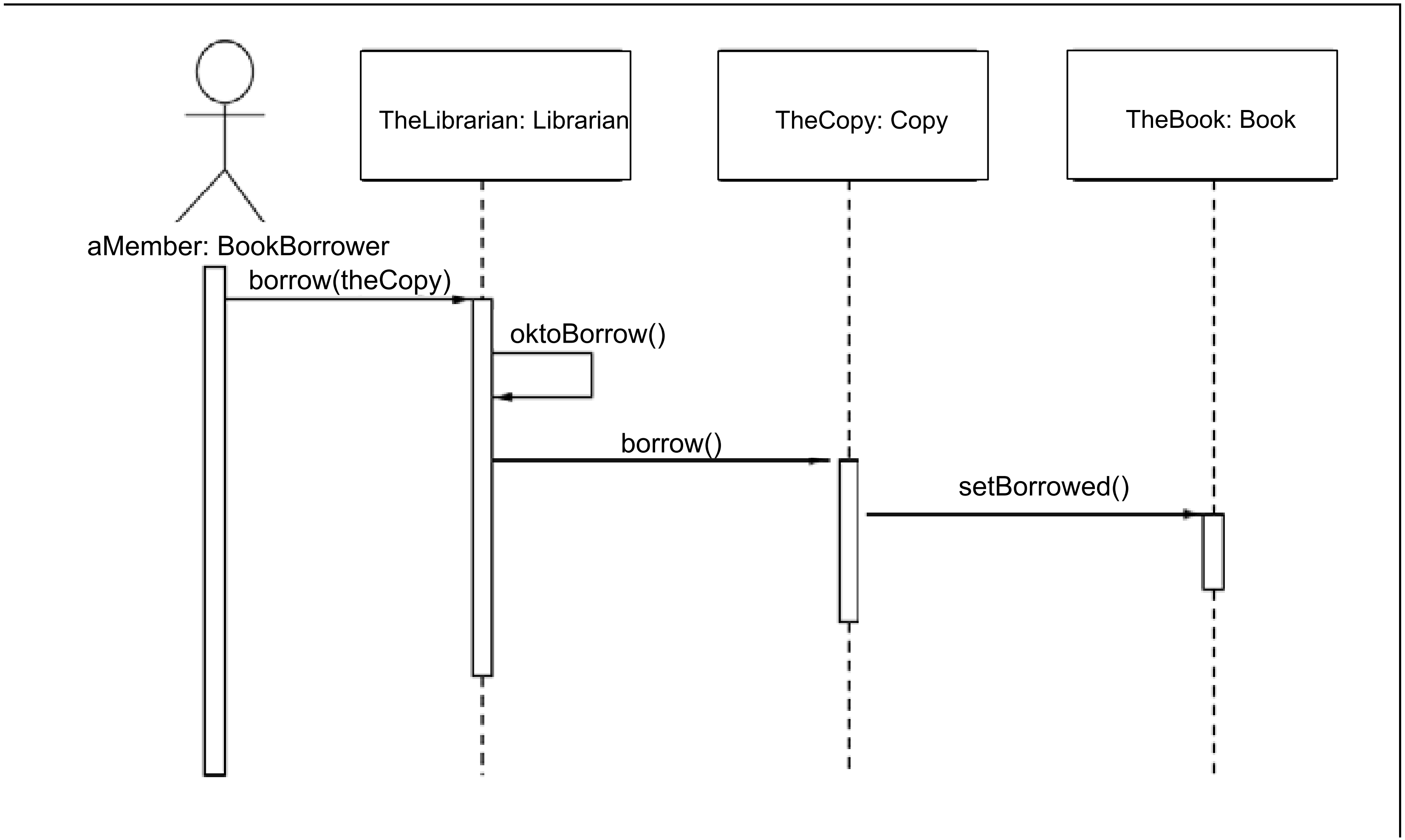
Library system, three objects do the work we're concerned with

BookBorrower: that will borrow the book

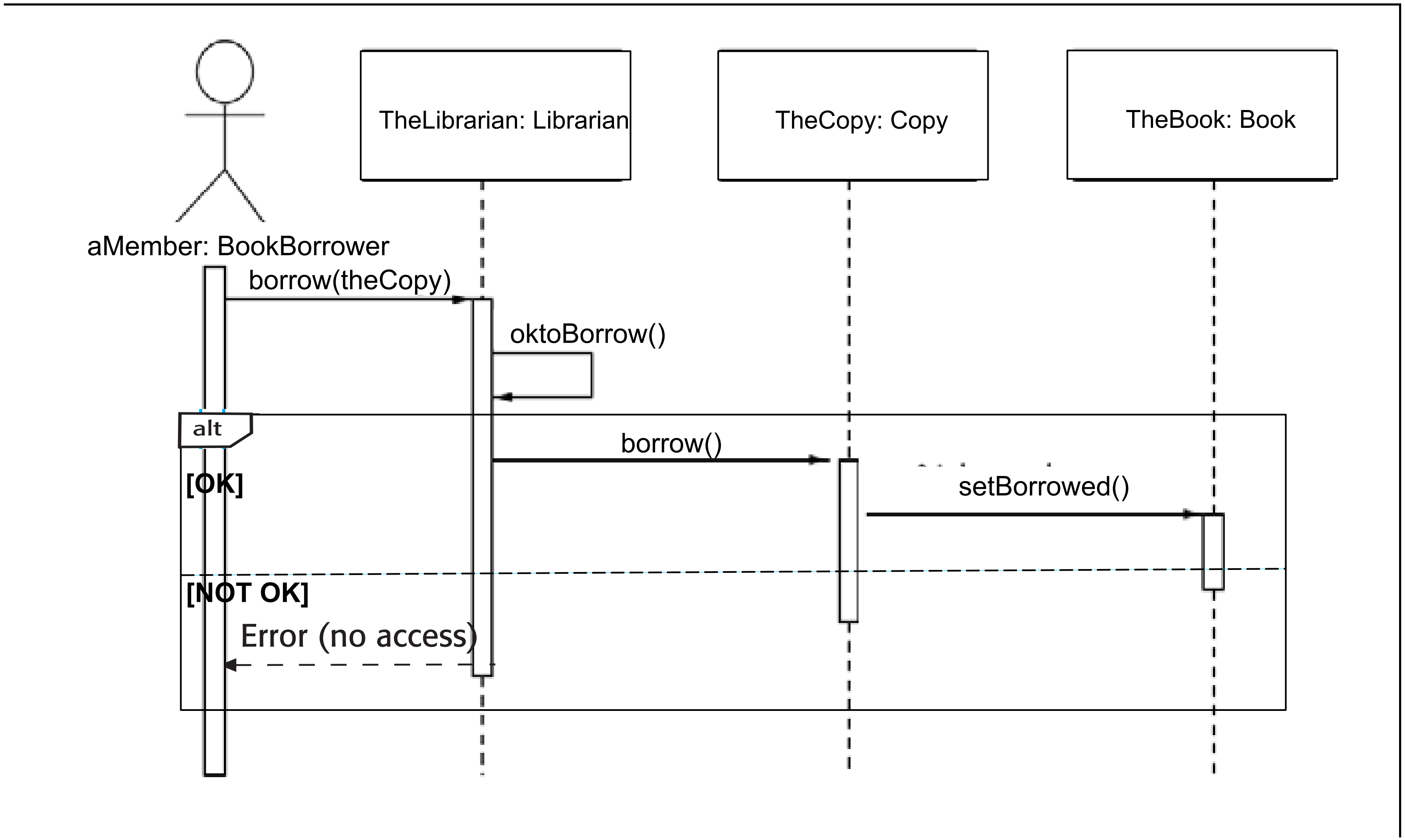
Copy: copy of a book

Librarian/LibraryStaff: which authorizes and register the borrowing of the borrowed copy.

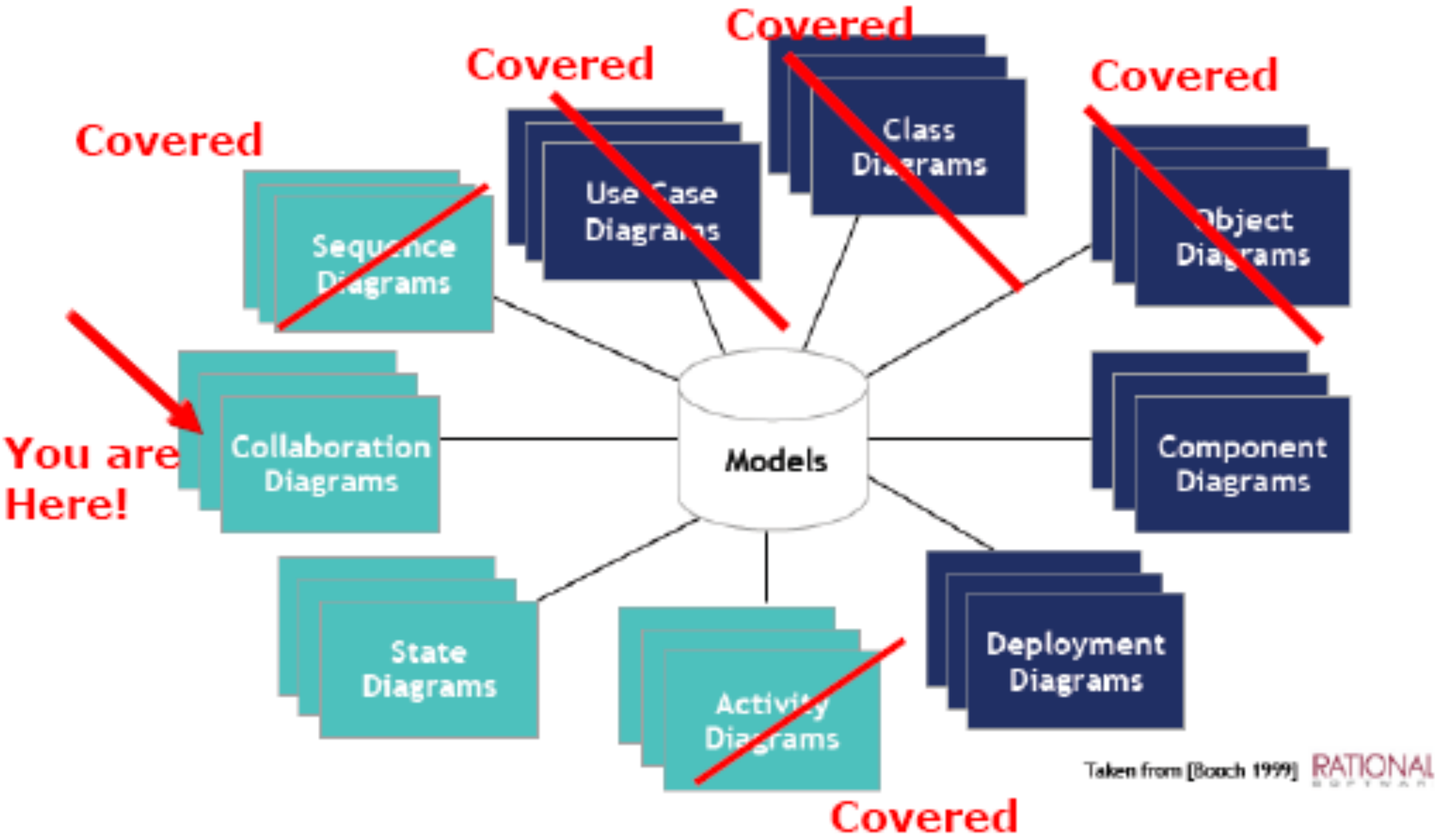
Sequence Diagram of a Library System



Sequence Diagram of a Library System



UML Diagrams



Taken from [Booch 1999]

Collaboration diagrams

Describe a specific scenario by showing the movement of messages between the objects

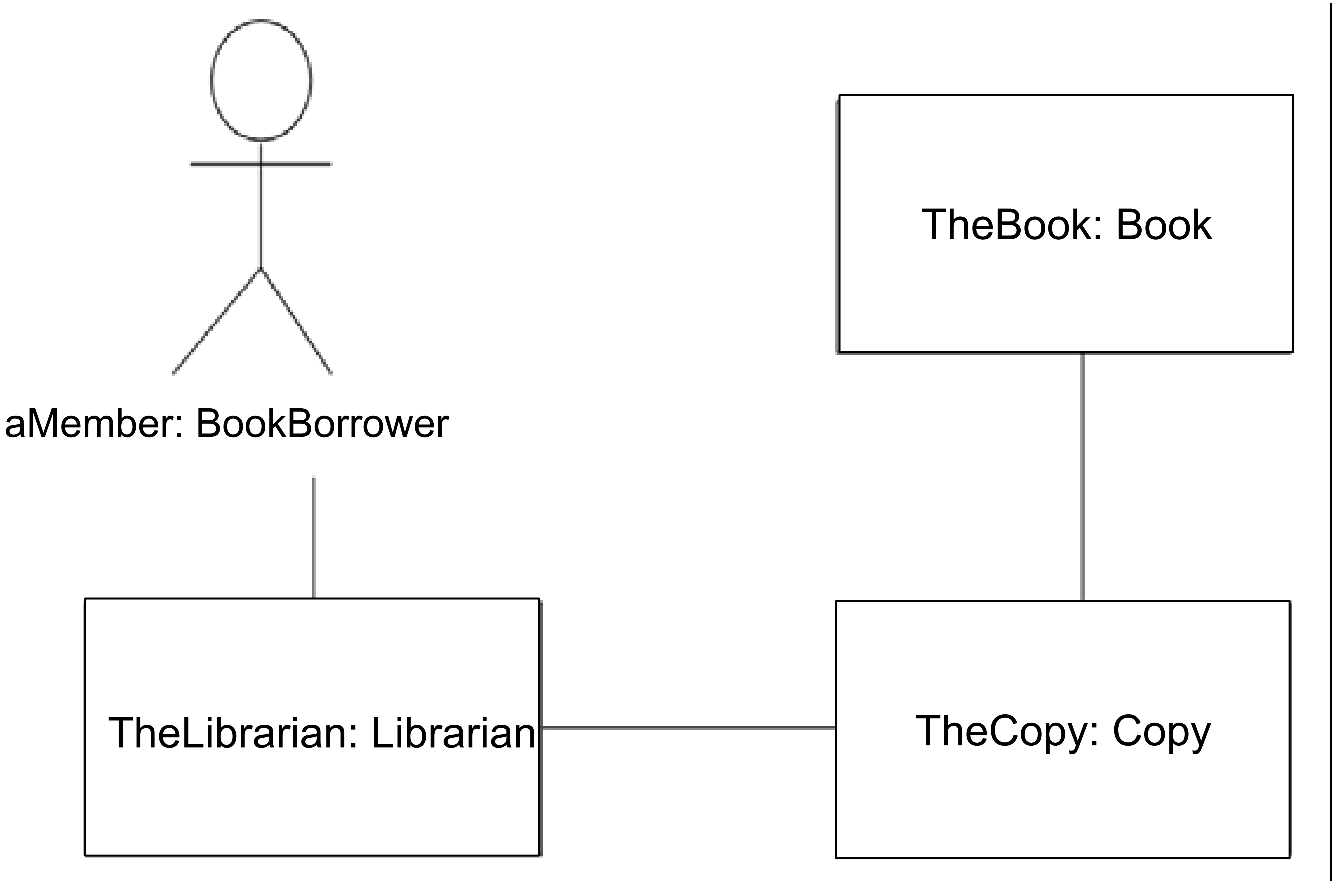
Show a spatial organization of objects and their interactions, rather than the sequence of the interactions

Unlike a Sequence diagram, a collaboration diagram shows the relationships among the objects. A collaboration diagram does not show time (i.e., sequence)

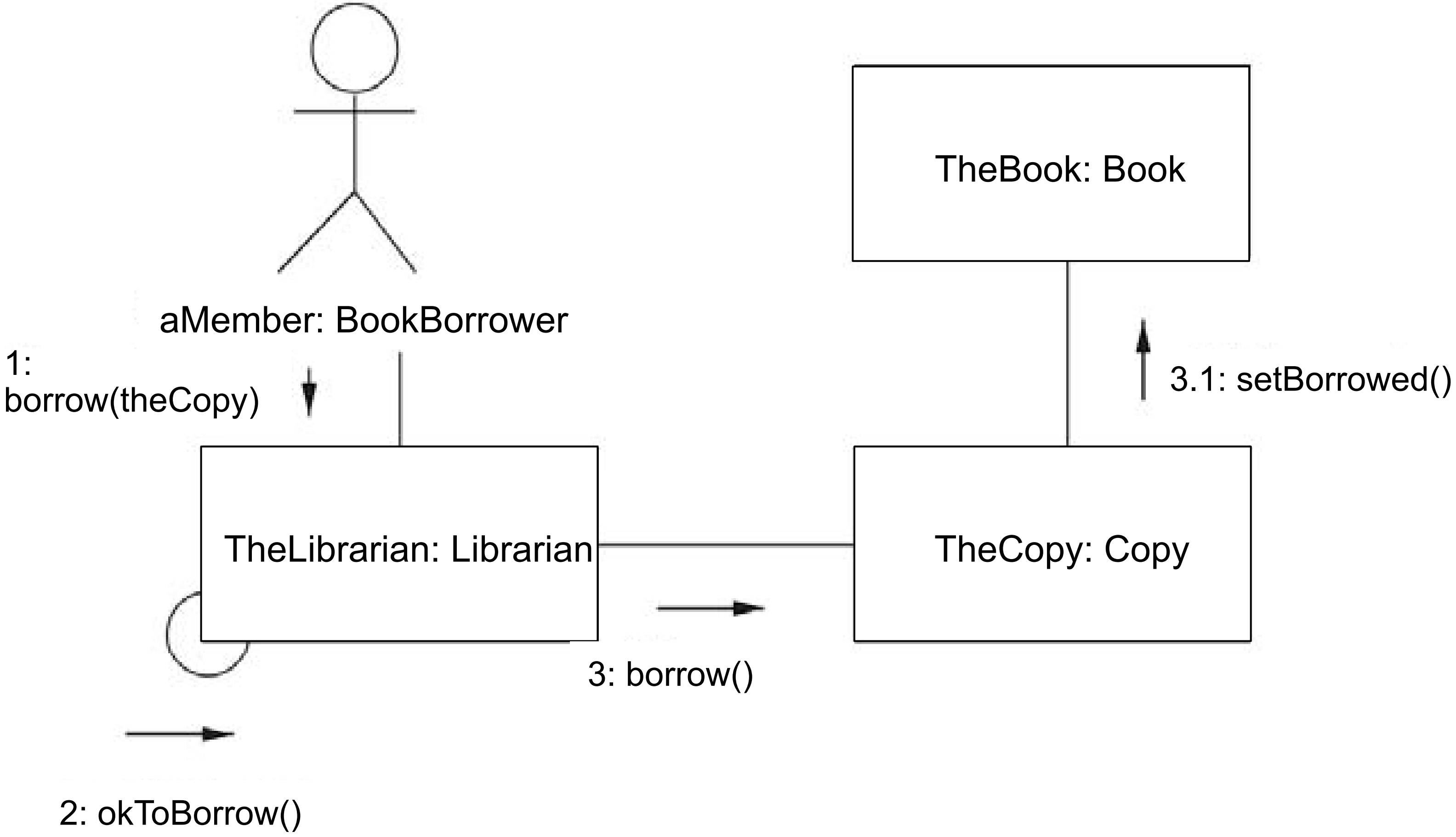
Keep in mind:- Both are referred to as interaction diagrams but with different focus!

- Sequence diagrams – models message flows between objects based on time (i.e., sequence)
- Collaboration diagrams– models message flows between objects with no reference to timing

Example- 1st:connect objects



Second: Draw interactions



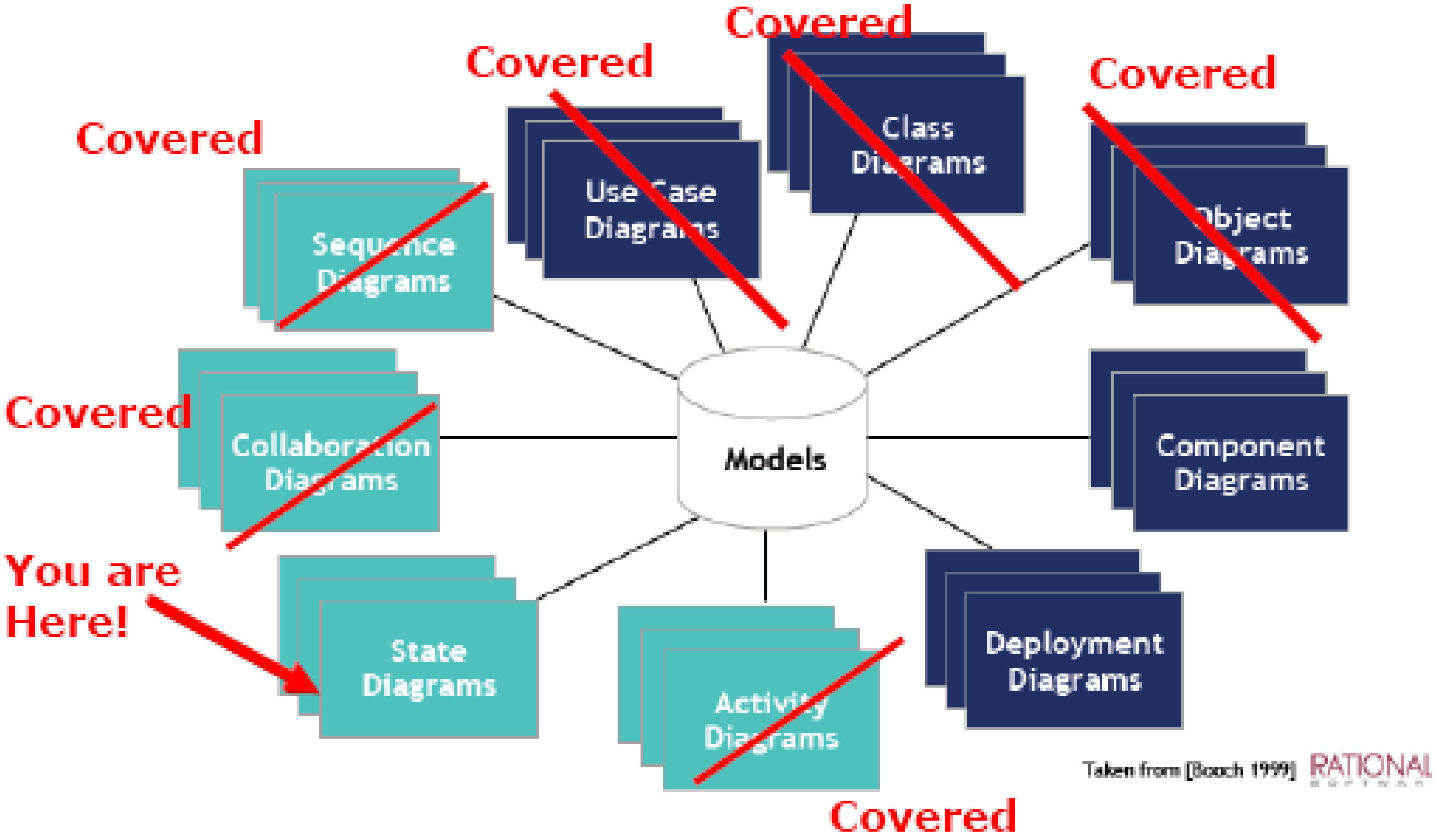
Exercise

Sketch a collaboration diagram for self-service machine, three objects do the work we're concerned with

- **the front:** the interface the self-service machine presents to the customer
- **the money register:** part of the machine where money is collected
- **the dispenser:** which delivers the selected product to the customer

Compare your collaboration diagram with that of a sequence diagram

UML Diagrams



Taken from [Booch 1999]

State Diagrams

Also known as statecharts (invented by David Harel)

Used primarily to model state of an object

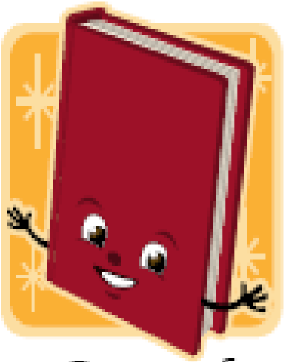
A class has at most one state machine diagram

- Models how an object's reaction to a message depends on its state
  - Objects of the same class may therefore receive the same message, but respond differently!

Use of State diagrams

Often used for modelling the behaviour of components (subsystems) of real time and critical systems….

Modelling states and events

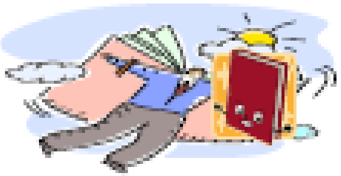


Copy of a Book

The **states** of the Book could be

On shelf

On loan



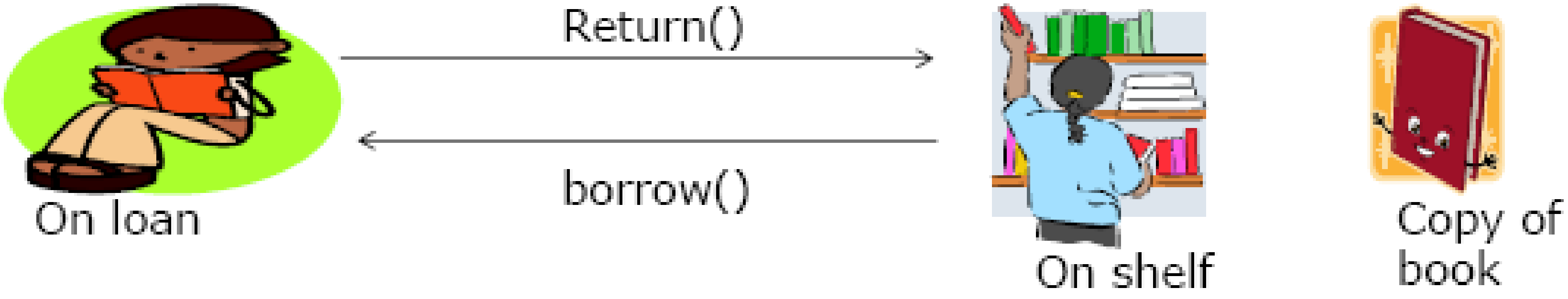
maybe lost

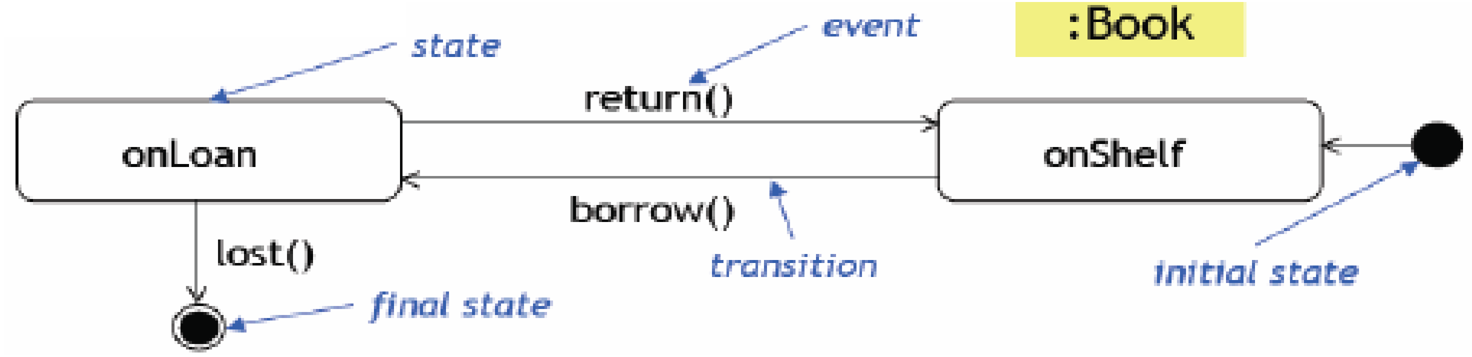
The related "**use cases**" or **events** could be

Borrow

return

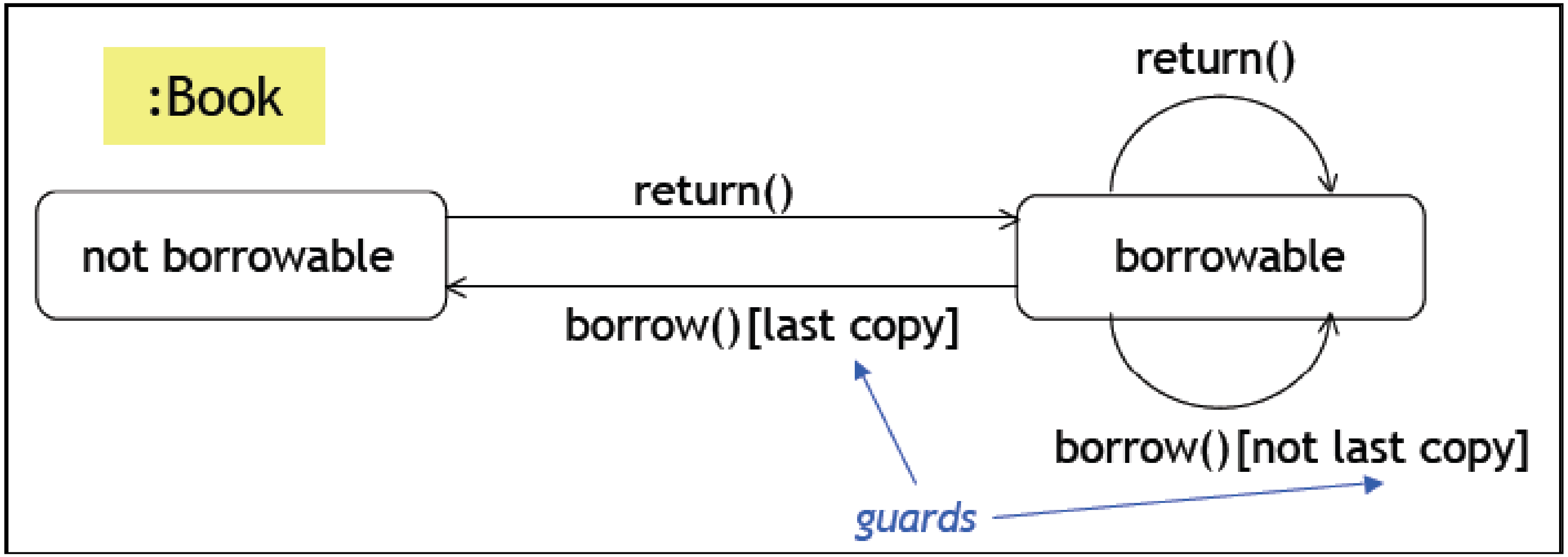
Realising state diagrams





Conditional notions

Conditional notation is used if the value of an object's attributes determines the change of state(i.e., change the state under this condition….)



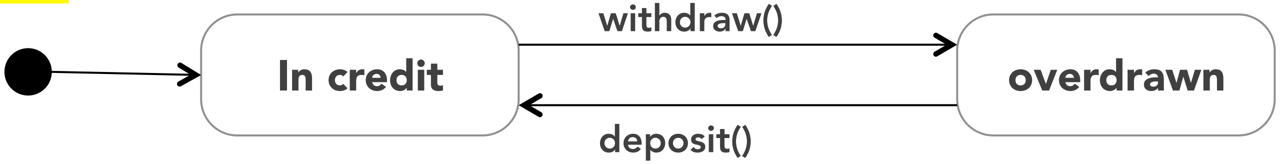
Important hint: For some *guards/conditions* use keywords like

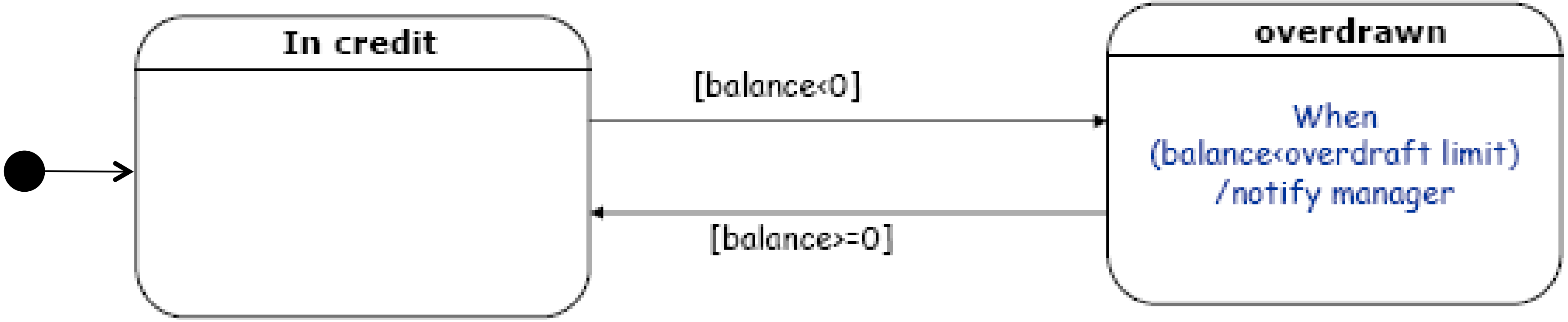
After (followed by expression)

When (followed by expression)

Conditional Notions

:BankAccount





Means...... when the withdraw()/deposit() use cases (or their corresponding methods) are invoked, then

If balance<0, then change the state to overdrawn

If balance>=0, then change the state to in-credit

Important hint:

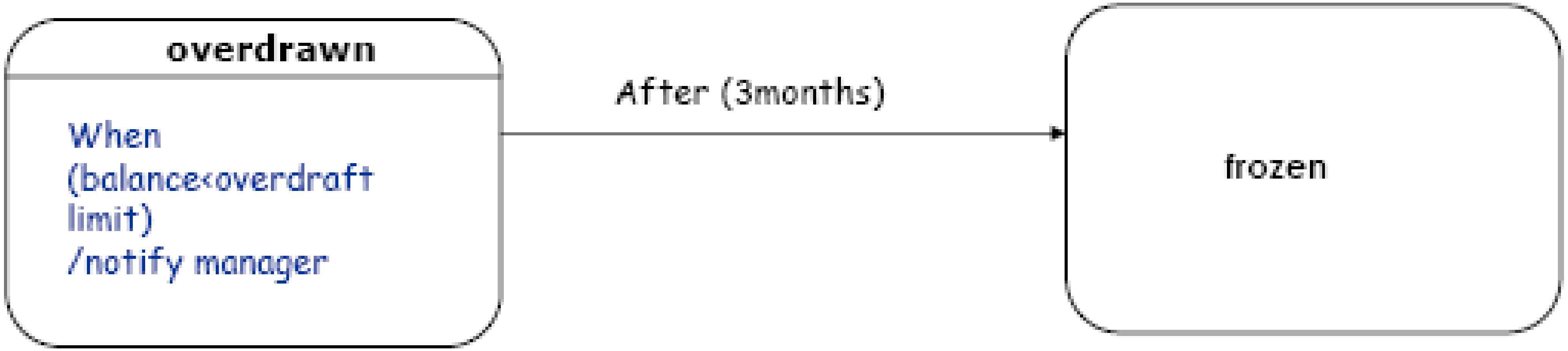
For expressing some events use keywords like

After (followed by expression)

When (followed by expression)

Conditional Notions

:BankAccount

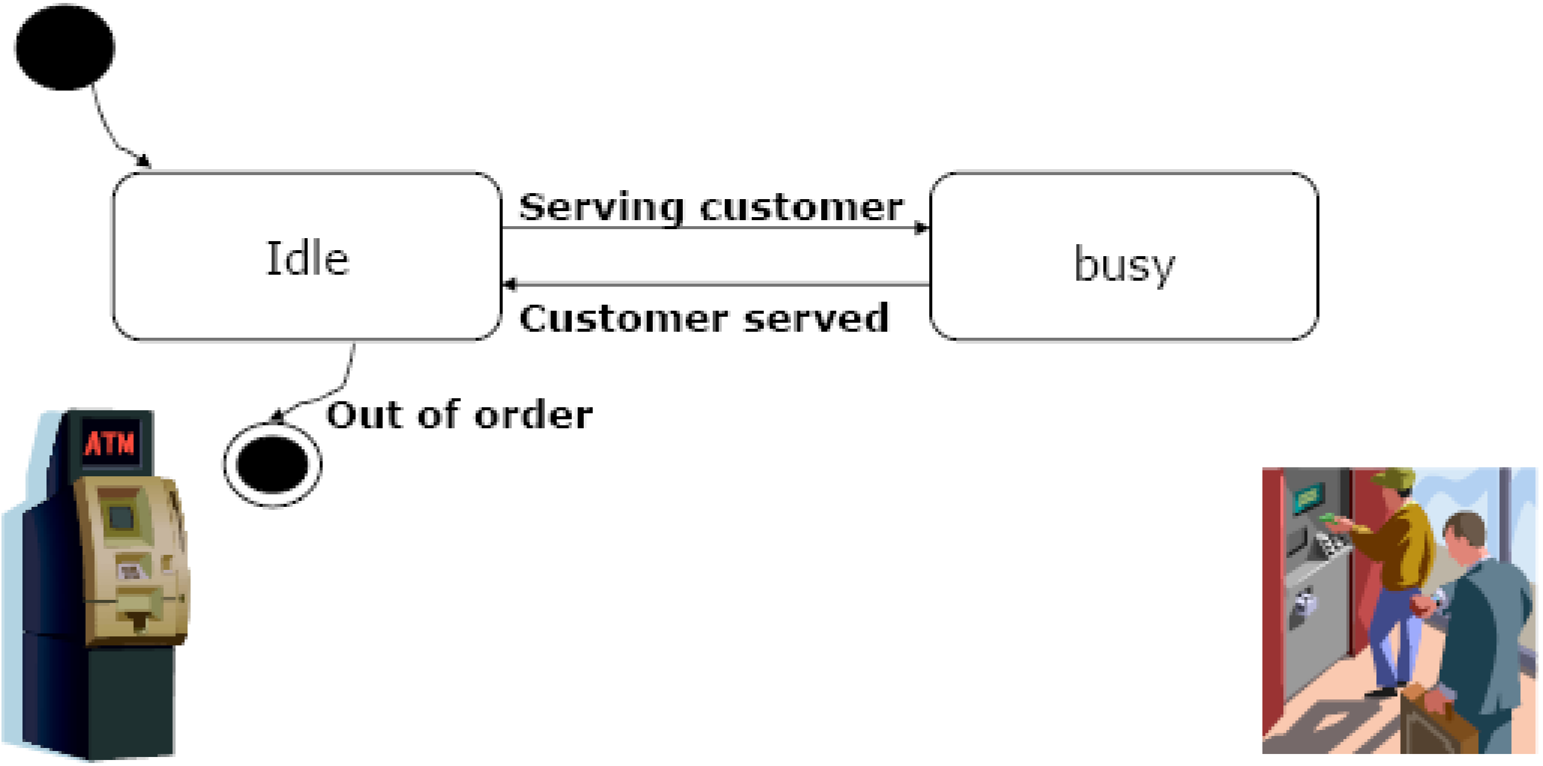


Important hint:
For expressing some events use keywords like

After (followed by expression)
When (followed by expression)

Modelling states and substates

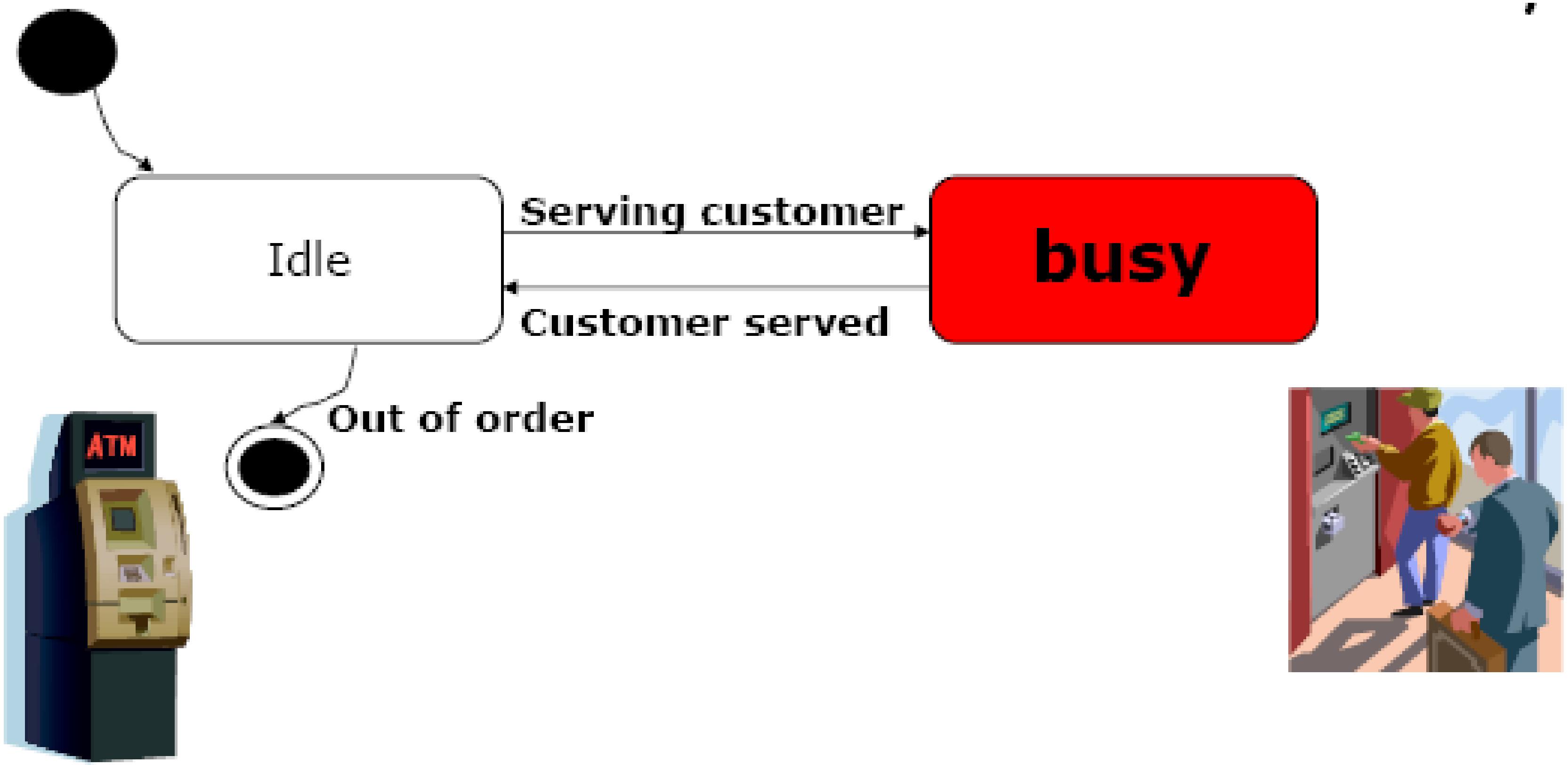
States of ATM machine itself…



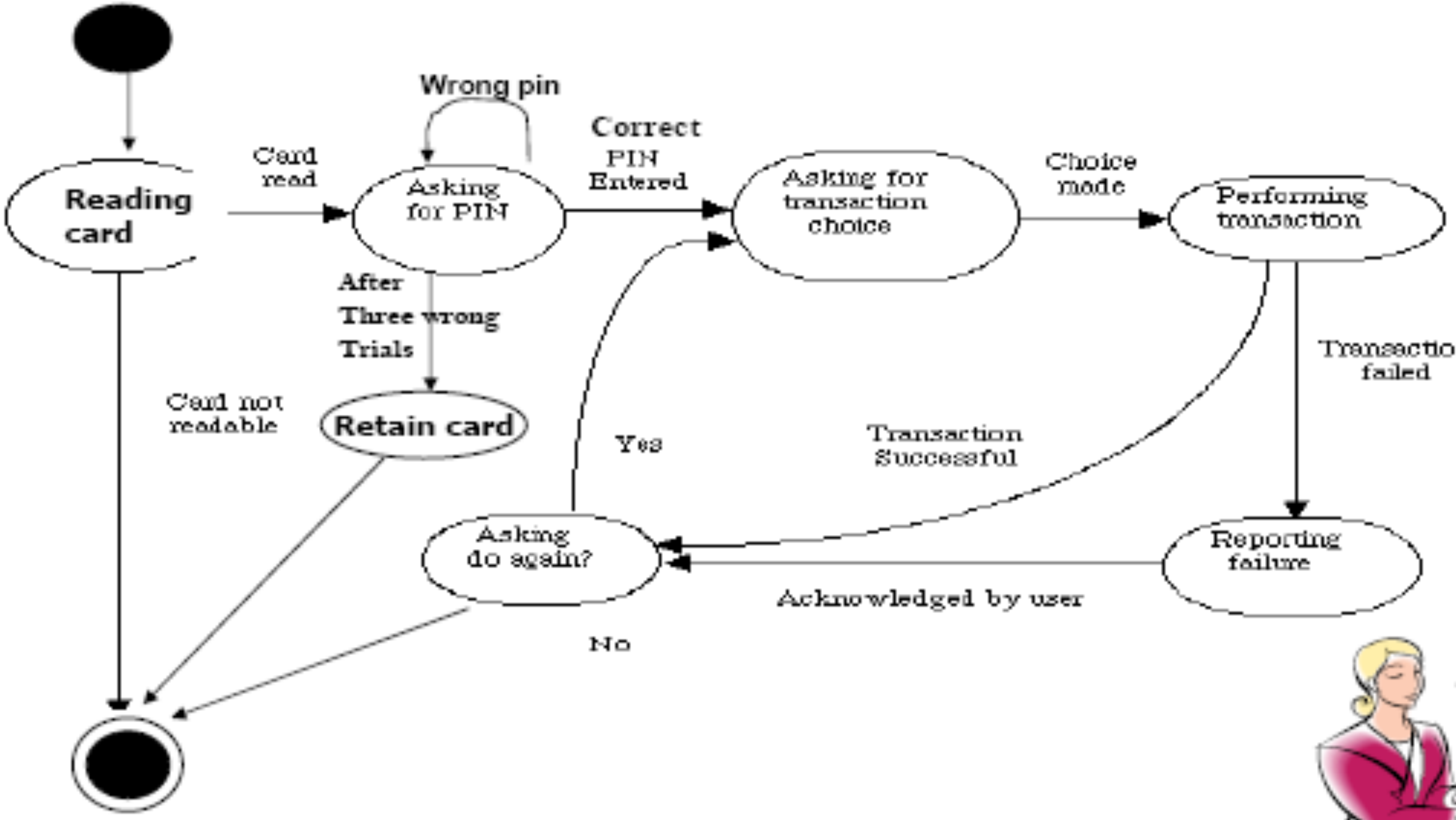
Modelling substates

States of ATM machine itself… are rather trivial!

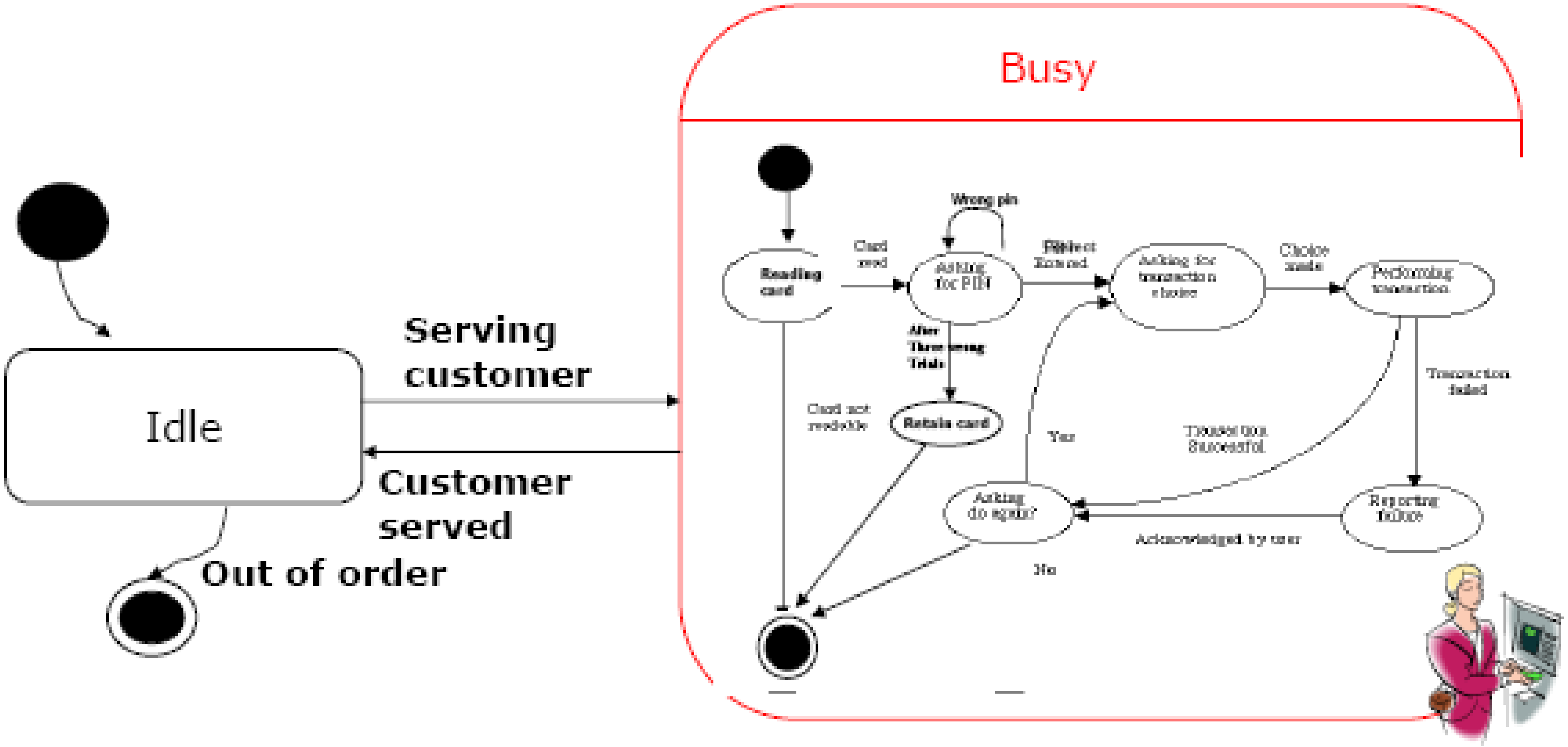
But useful to model the composed state busy to create its sub states to understand more fully the ATM states for a developer to implement.



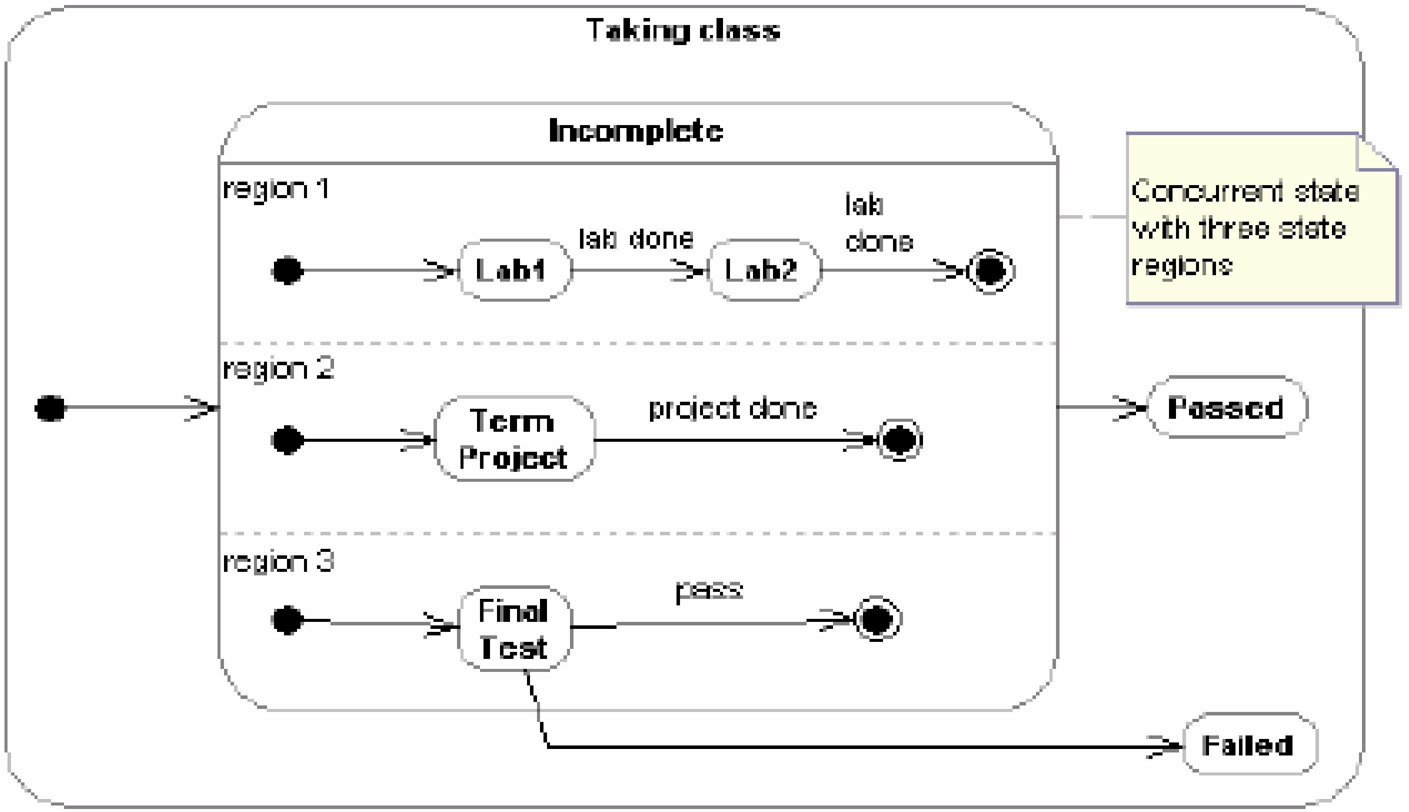
Modelling substates of ATM machine



Modelling substates of ATM machine



Modelling concurrent states



Concurrent state with three state regions

States that occur in parallel

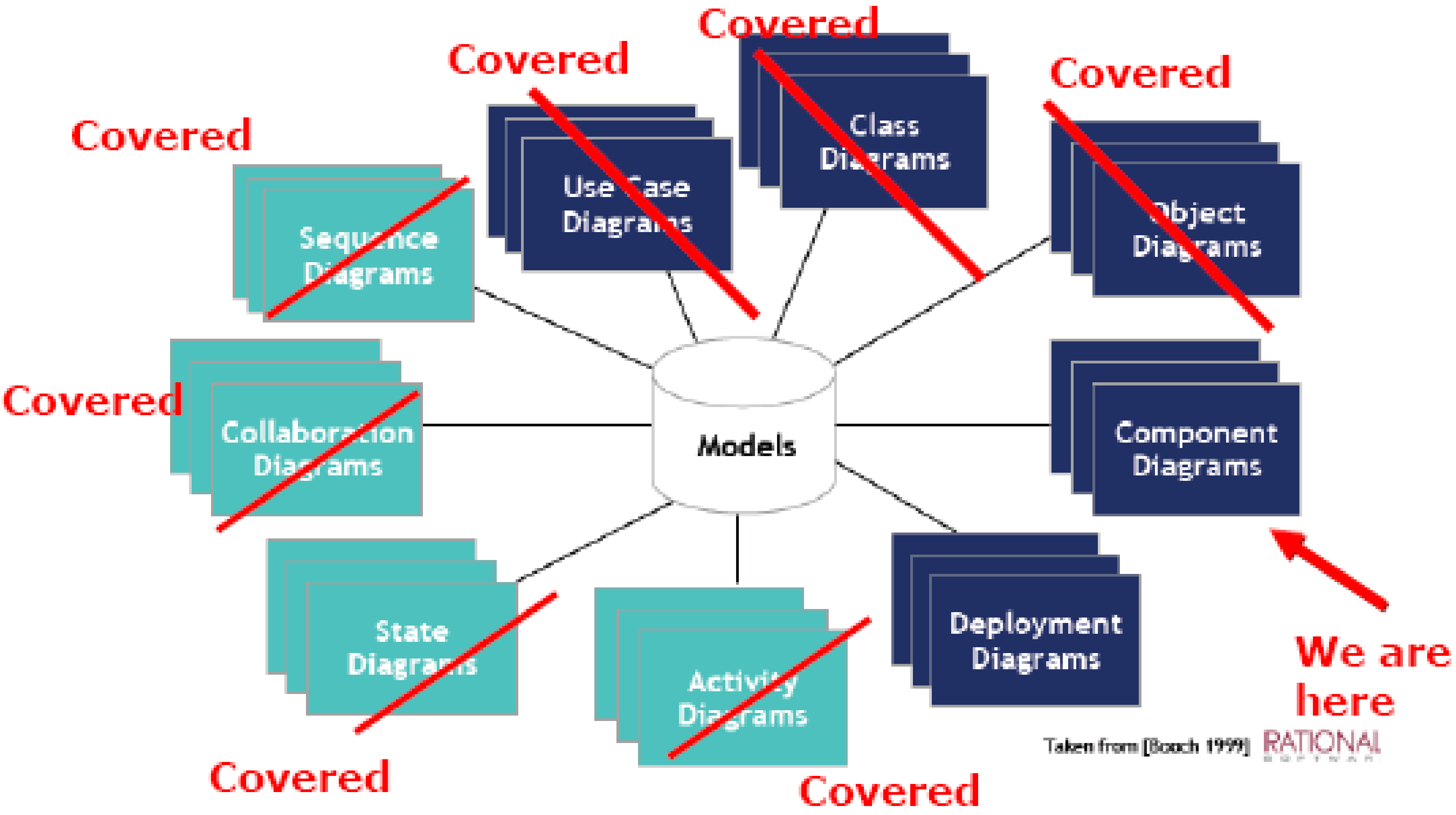
Exercise: a State diagram of a video player

- What are the states of the player?
- What are the events that cause state changes?
- What are the outputs that occur?
- What are the guards for the transitions?

Reference: David Rosenblum, UCL

- What would we model differently in an activity diagram for the player?

UML Diagrams



Component Diagrams

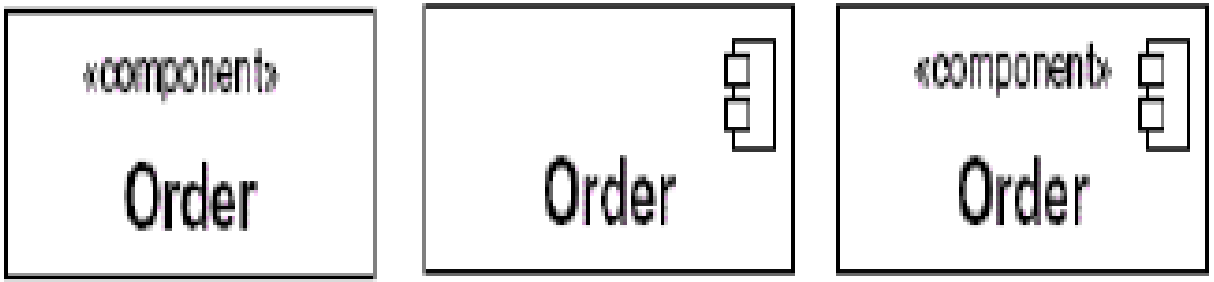
The component diagram's main purpose is to show the structural relationships between the components of a system

Component diagrams offer architects a natural format to begin modelling a solution

Component diagrams allow an architect to verify that a system's required functionality is being implemented by components

Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building

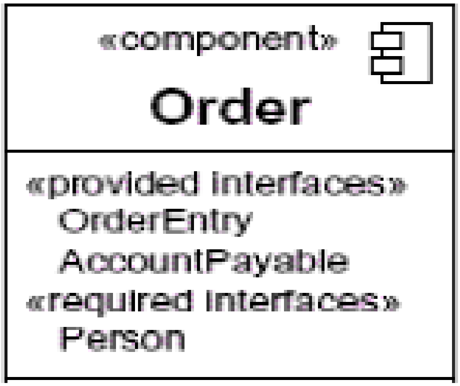
Component Diagrams



All they mean the same: a component Order

UML version 2.0

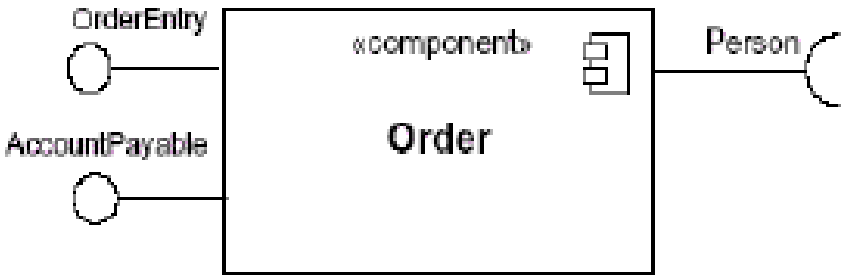
Required/Provide Interface



«component»

Order

«provided interfaces»
OrderEntry
AccountPayable
«required interfaces»
Person



OrderEntry

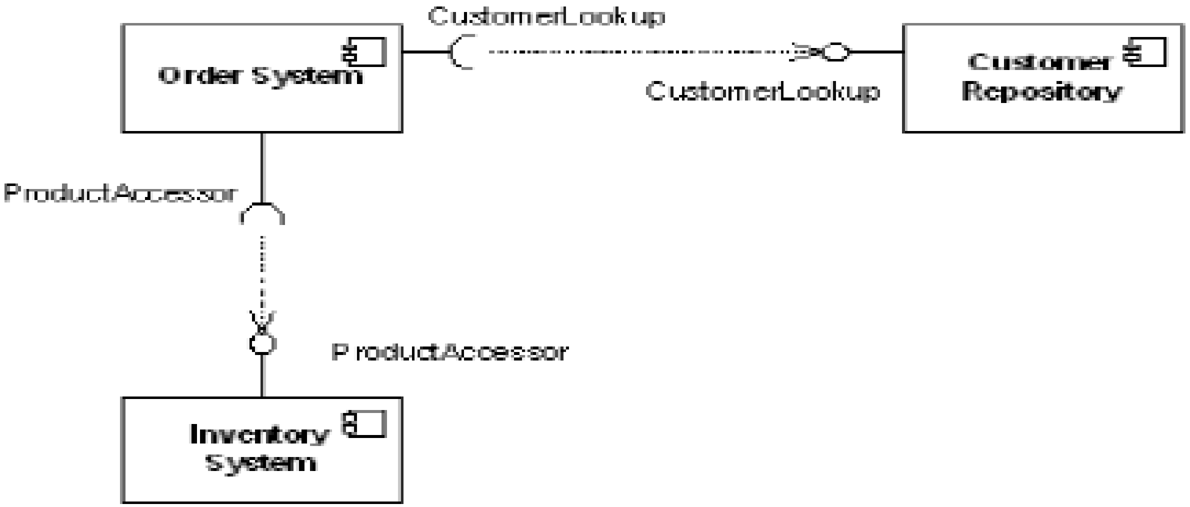
AccountPayable

«component»

Order

Person

Component Diagrams

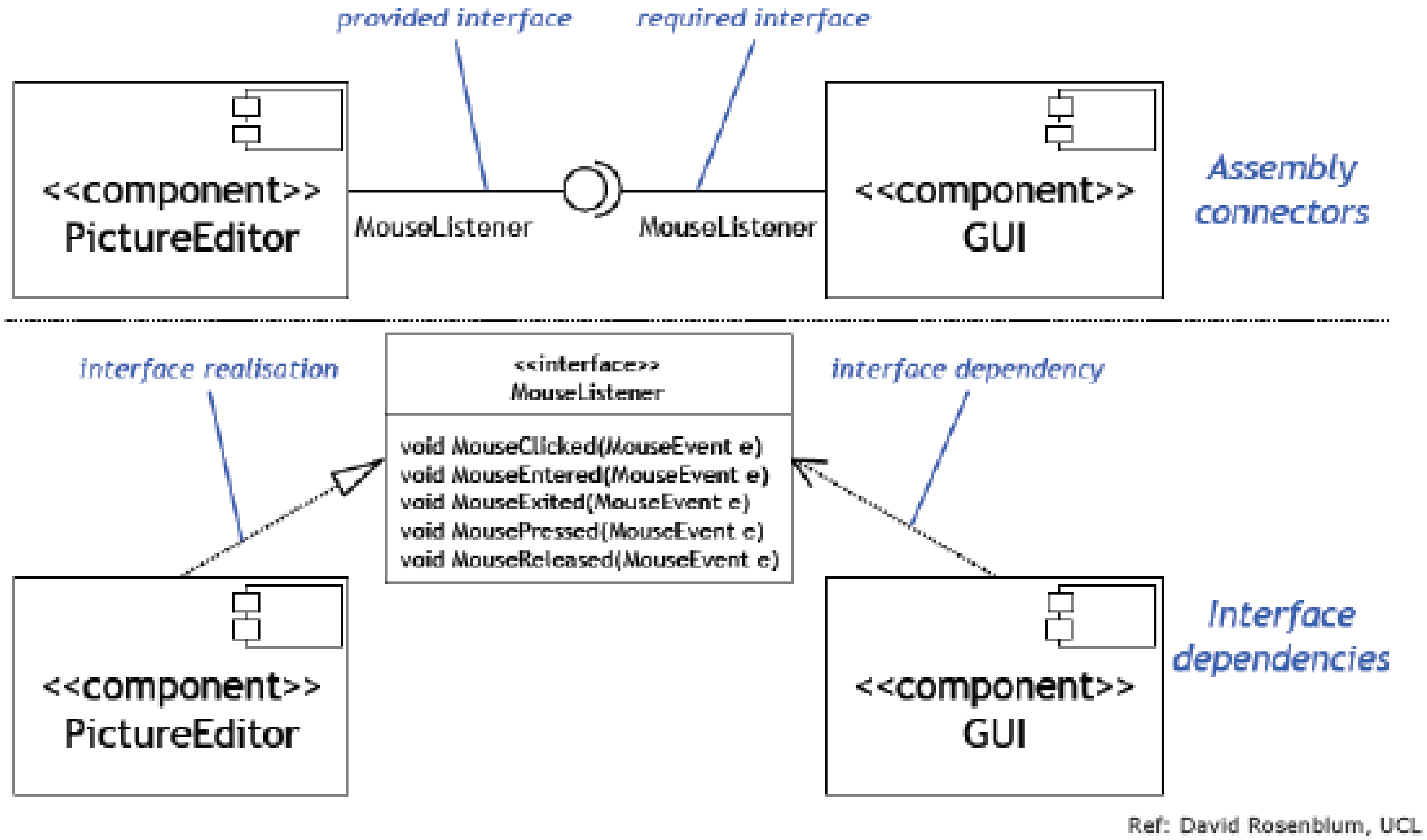


showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop

Component Diagrams

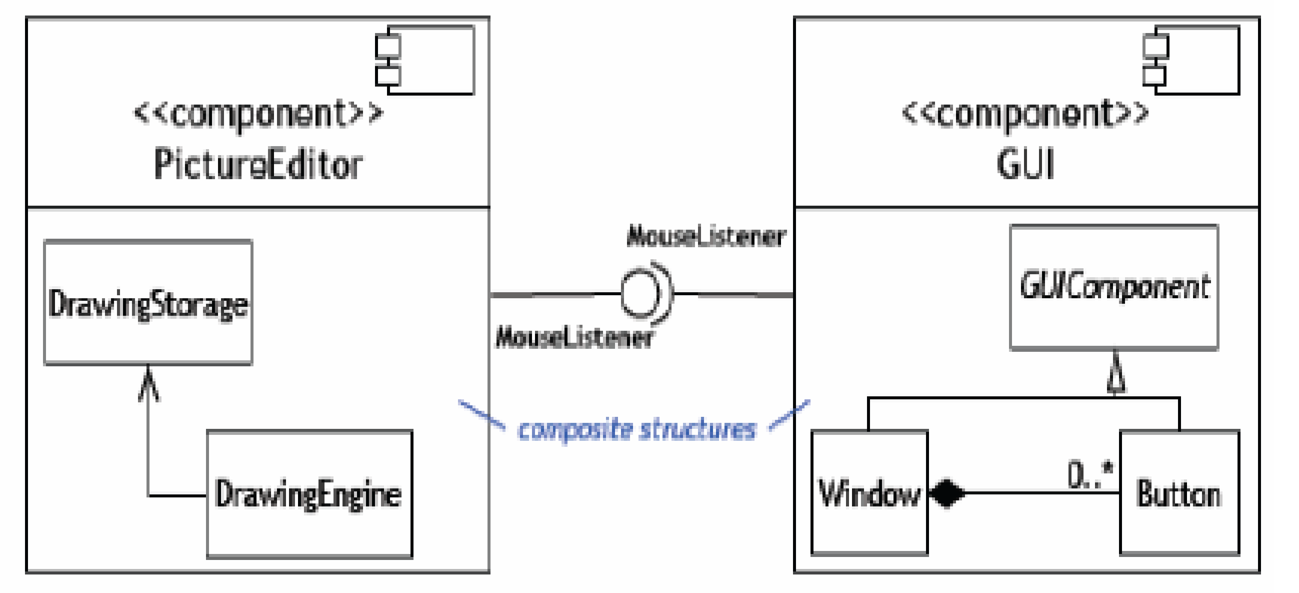
- Architectural **connection** in UML 2.0 is expressed primarily in terms of interfaces
- Interfaces are classifiers with operations but no attributes
- Components have **provided** and **required interfaces**
  - Component implementations are said to **realize** their provided interfaces
  - A provided and required interface can be connected if the operations in the latter are a subset of those in the former, and the signatures of the associated operations are ‘**compatible**’
- **Ports** provide access between external interfaces and internal structure of components
- UML components can be used to model complex architectural connectors (like a CORBA ORB)

Component Diagrams



Ref: David Rosenblum, UCL

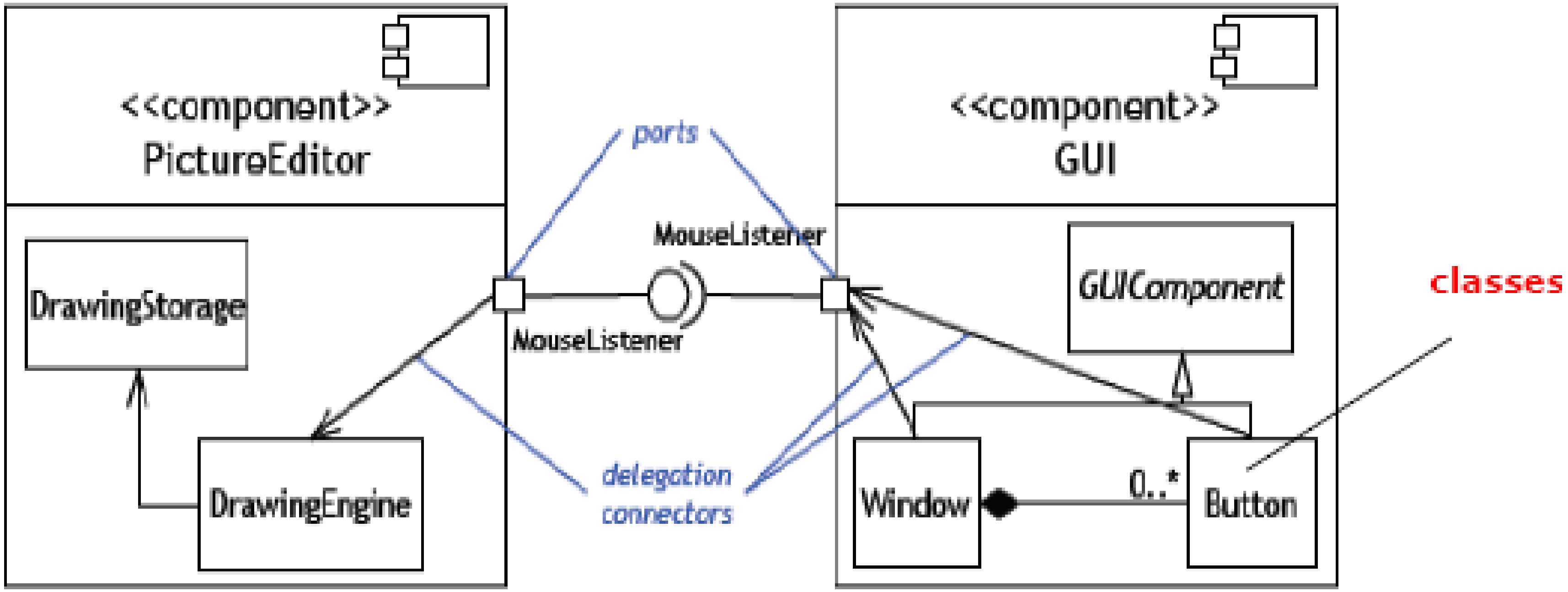
Composite Structure in Component Diagrams



Ref: David Rosenblum, UCL

A composite structure depicts the internal realisation of component functionality

Ports

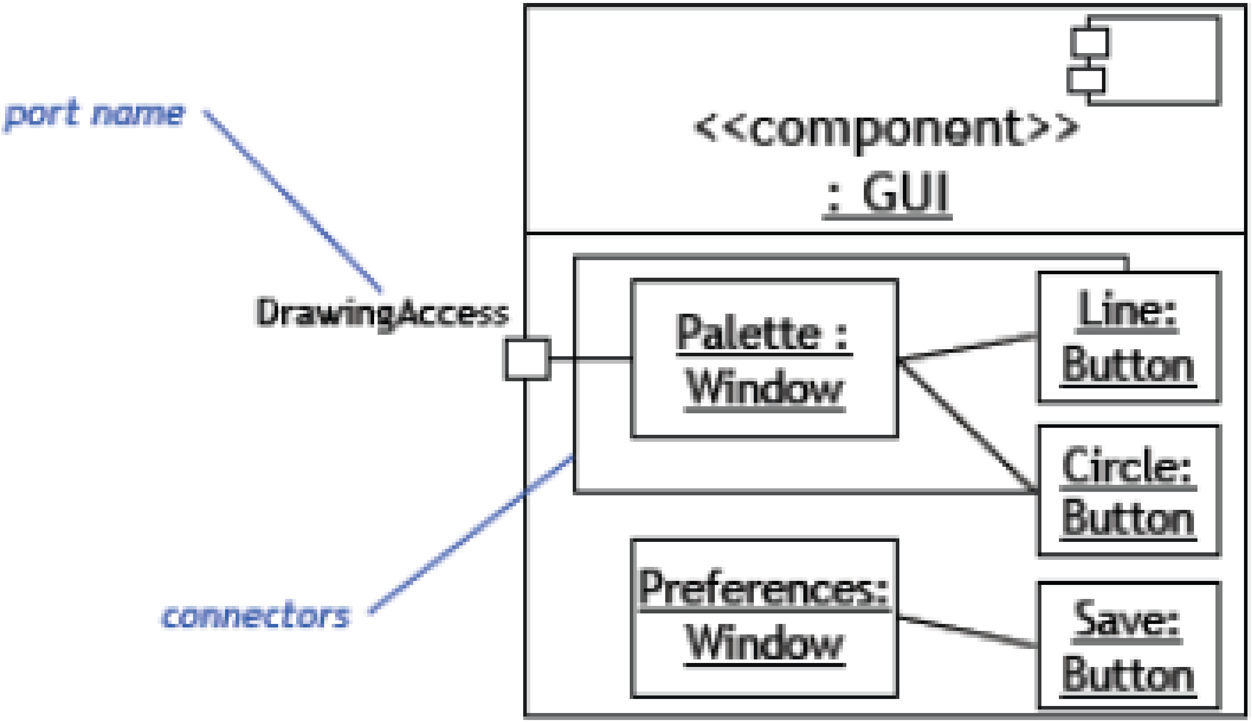


Ref: David Rosenblum, UCL

The ports and connectors specify how component interfaces are mapped to internal functionality

Note that these 'connectors' are rather limited, special cases of the ones in software architectures

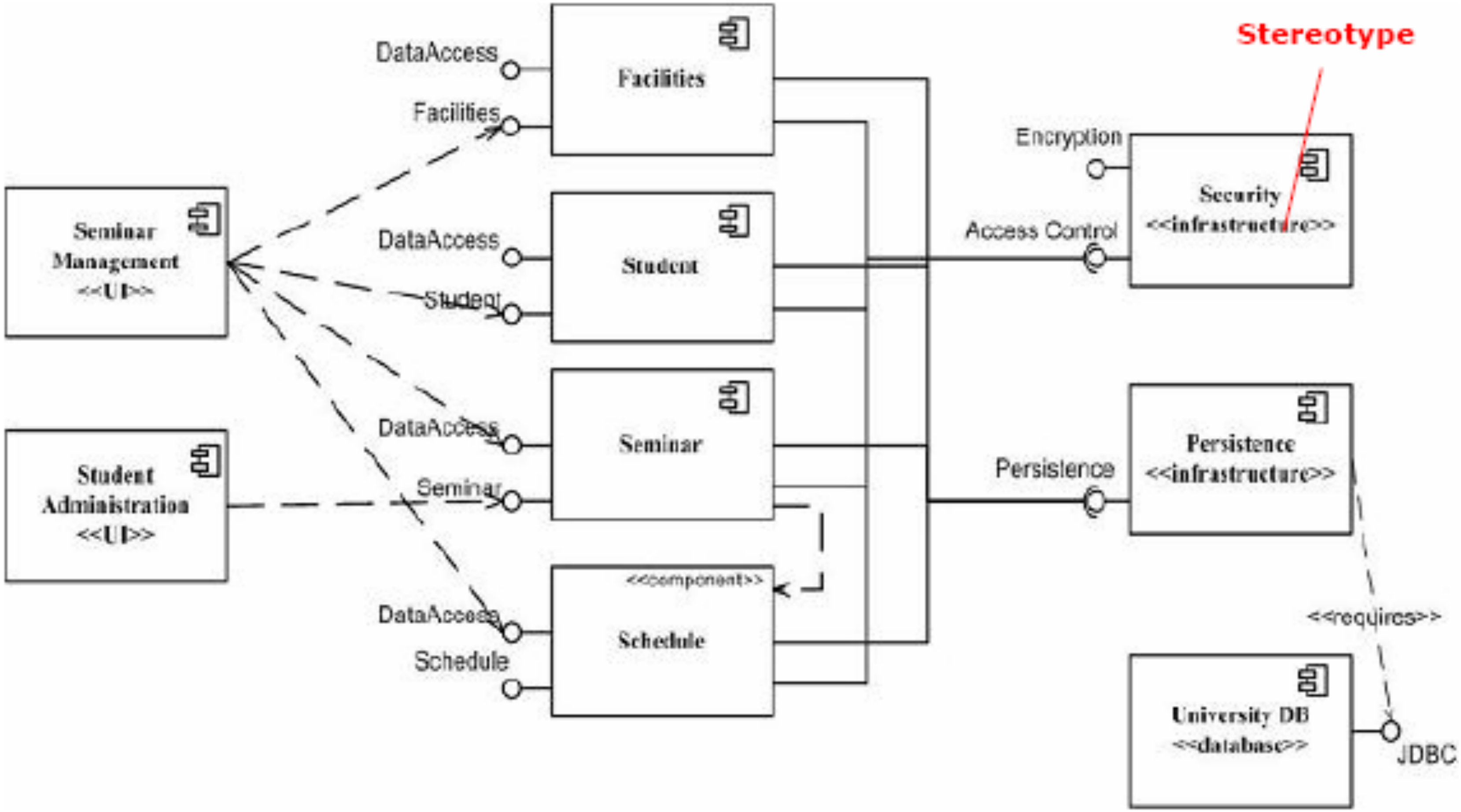
Ports



Ref: David Rosenblum, UCL

Connectors and ports also can be used to specify structure of component ***instantiations***

Example


Stereotype
DataAccess
Facilities
Facilities
Seminar Management <<UI>>
DataAccess
Student
Student
DataAccess
Seminar
Seminar
Student Administration <<UI>>
<<component>>
DataAccess
Schedule
Schedule
Encryption
Access Control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
<<requires>>
University DB <<database>>
JDBC

Componentization Guidelines

"Keep components *cohesive*". i.e a component should implement a single, related set of functionality.

This may be the user interface logic for a single user application, business classes comprising a large-scale domain concept, or technical classes representing a common infrastructure concept.

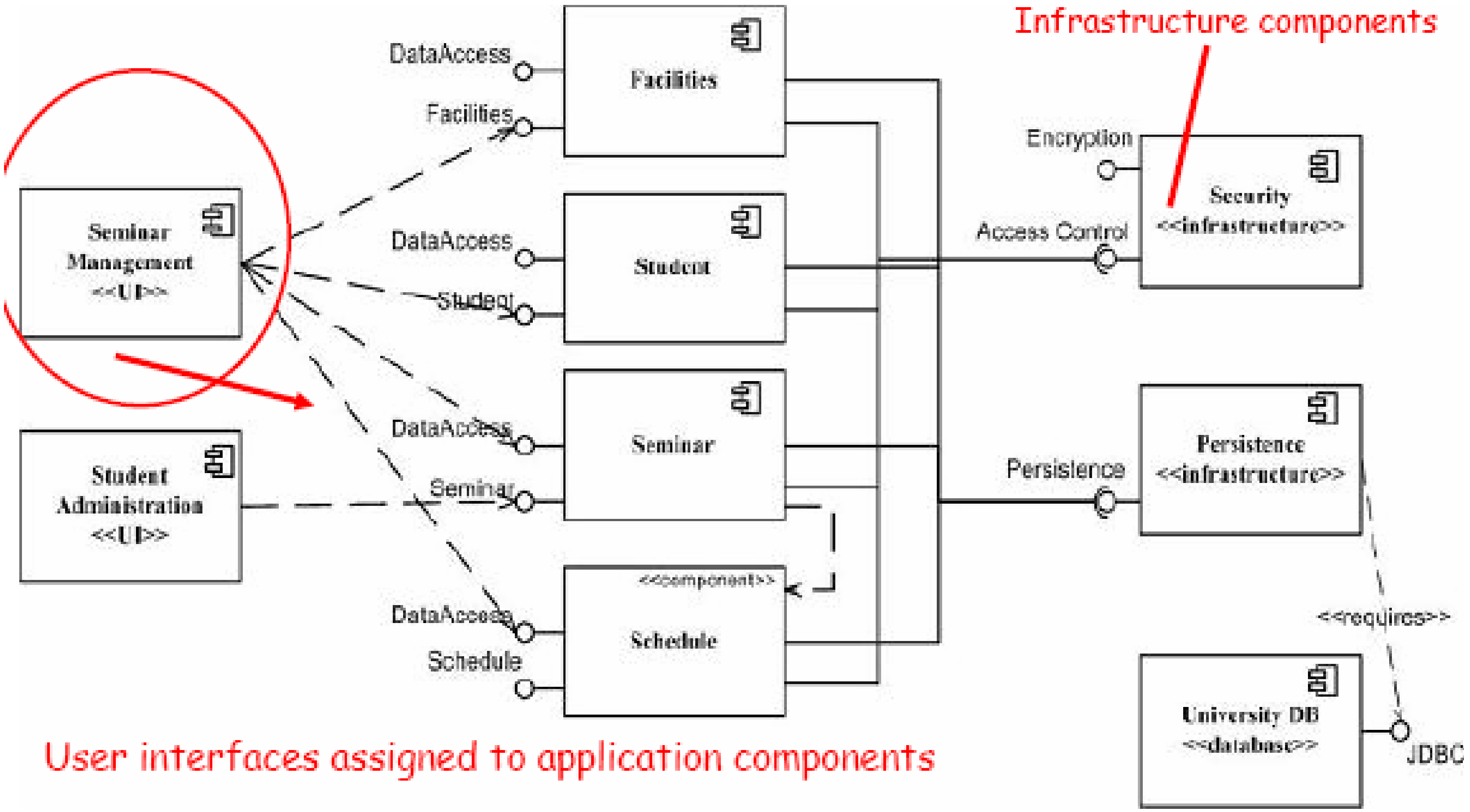
User *interface* classes assigned as application components.

User interface classes, those that implement screens, pages, or reports, as well as those that implement "glue logic".

Assign common technical classes to *infrastructure components*.

Technical classes, e.g. that implement system-level services such as security, persistence, or middleware should be assigned to components which have the *infrastructure stereotype*.

Example


Infrastructure components
DataAccess
Facilities
Facilities
Seminar Management
<<UI>>
DataAccess
Student
Student
Encryption
Security
<<infrastructure>>
Access Control
DataAccess
Seminar
Seminar
Student Administration
<<UI>>
Persistence
Persistence
<<infrastructure>>
<<component>>
DataAccess
Schedule
Schedule
<<requires>>
University DB
<<database>>
JDBC
User interfaces assigned to application components

Componentization Guidelines

Assign *hierarchies* to the same component.

99.9% of the time it makes sense to assign all of the classes of a hierarchy, either an *inheritance hierarchy* or a *composition hierarchy*, to the same component.

Identify business domain components.

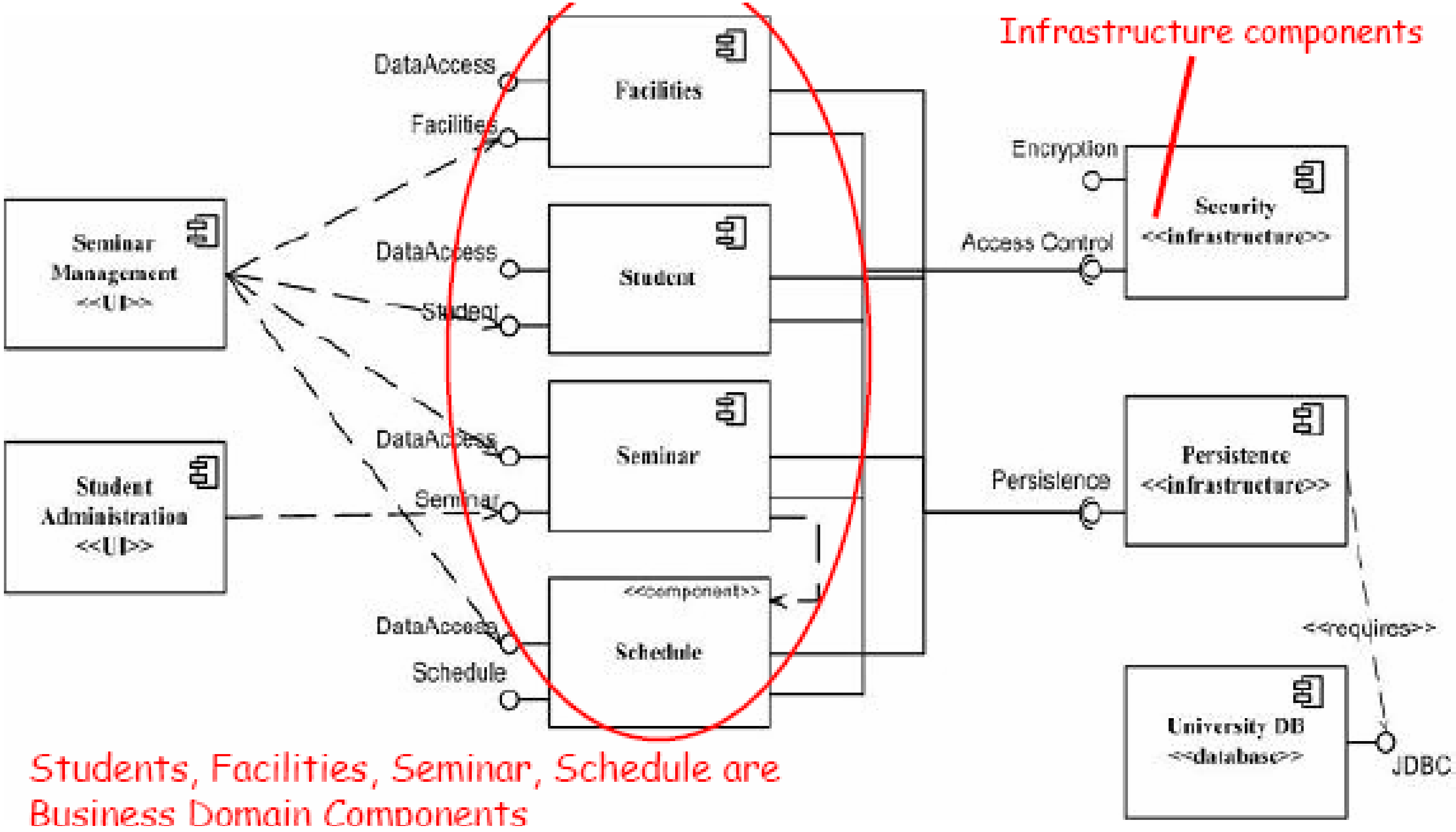
Because you want to minimize network traffic to reduce the response time of your application, you want to design your business domain components in such a way that most of the *information flow* occurs *within* the components and not *between* them.

Business domain components* = *business services

Identify the "collaboration type" of business classes.

Once you have identified the collaboration type of each class (e.g. server/client or both), you can start identifying potential business domain components.

Example



Infrastructure components

Students, Facilities, Seminar, Schedule are Business Domain Components

Componentization Guidelines

Highly coupled classes grouped in the same component.

When two classes collaborate frequently, this is an indication they should be in the same domain business component to reduce the network traffic between the two classes.

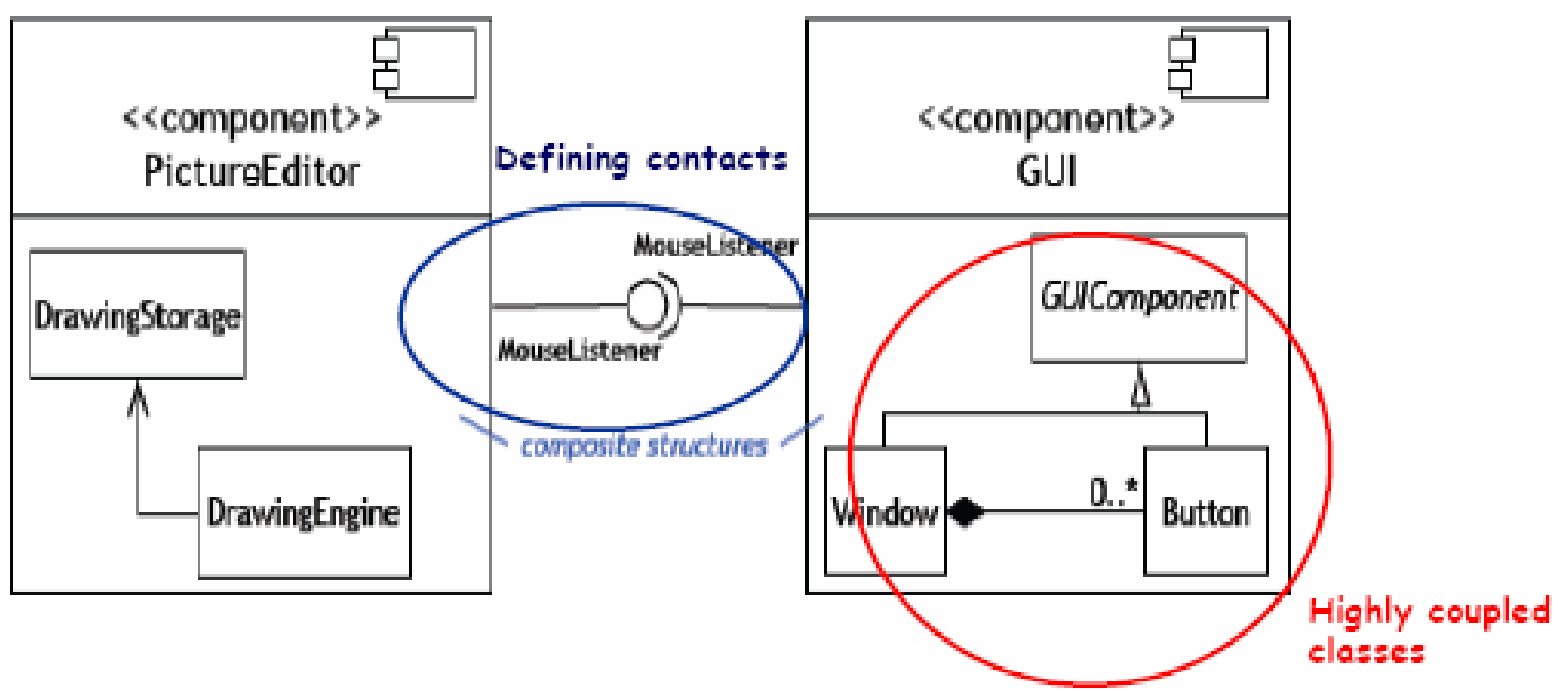
Minimize the size of the *message flow* between components.

If you have domain components, one as a server to only the other as a client, you may decide to combine or merge the two components.

Define component *contracts,* as interfaces.

Each component will offer services to its client components, each such service is a component contract.

Example



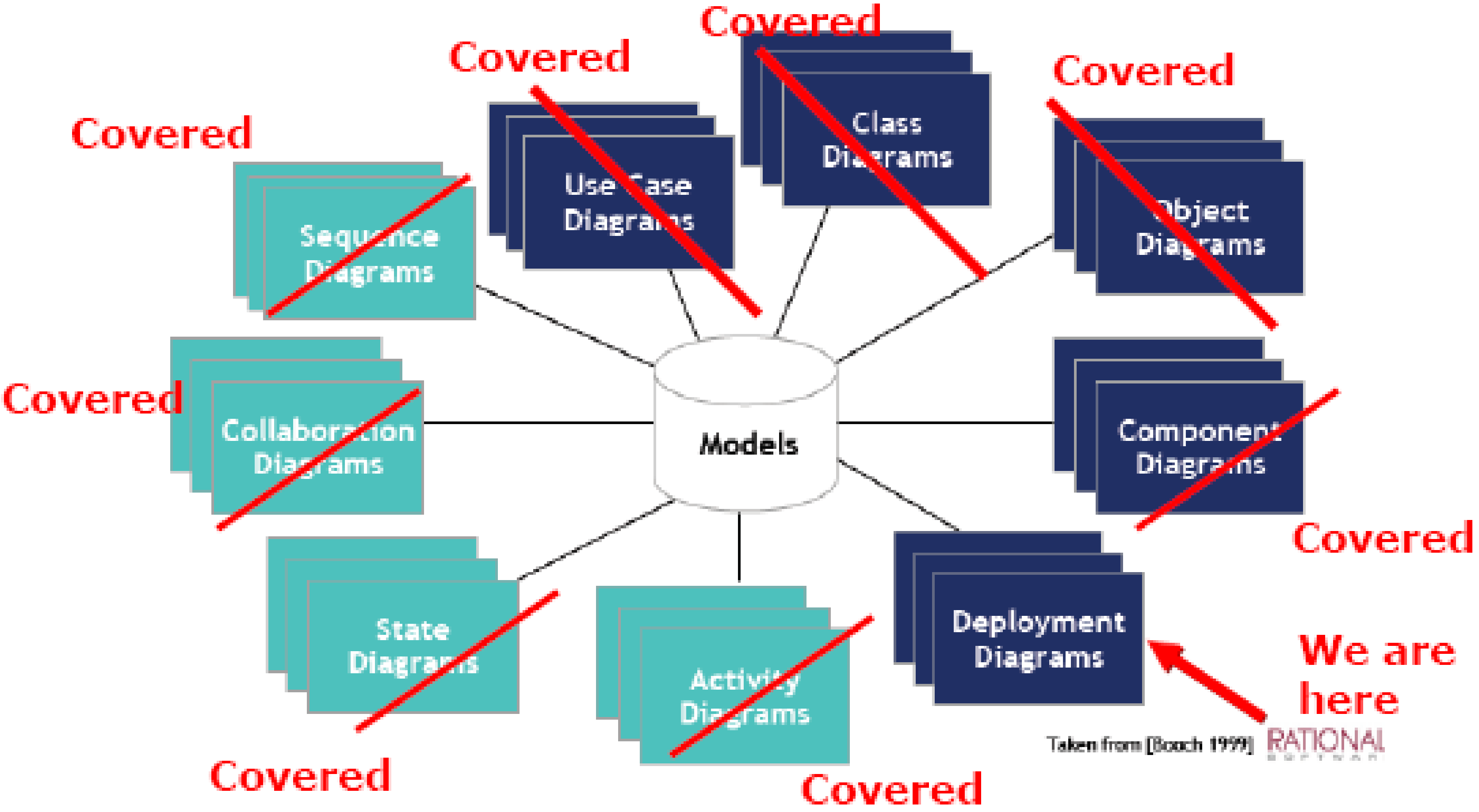
Highly coupled classes belong in the same component

Ref: David Rosenblum, UCL

Exercise

Draw a component diagram of an ATM machine

UML Diagrams



Taken from [Booch 1999]

Deployment Diagram

Models the *run-time* configuration in a static view and visualizes the distribution of components in an application

It helps map between software components and hardware

A component is deployed part of the *software system architecture*

In most cases, it involves modelling the *hardware* configurations together with the *software* components that lived on

Deployment Diagram

Deployment diagram depicts a *static view* of the run-time configuration of processing nodes and the components that run on those nodes

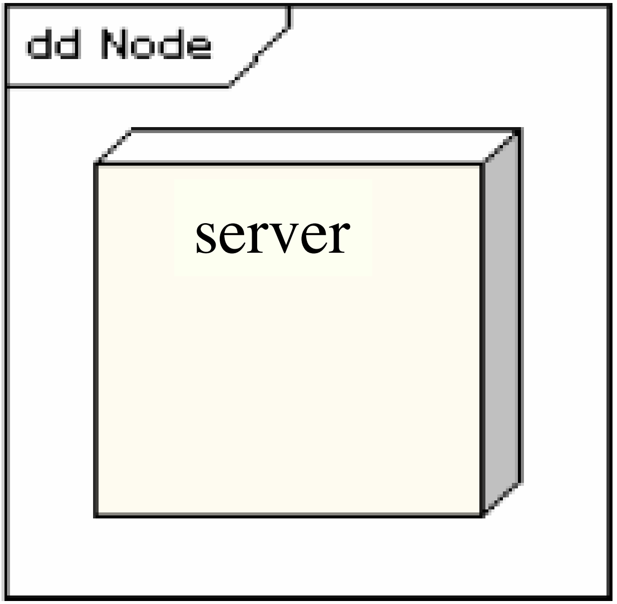
Node: server, client etc.

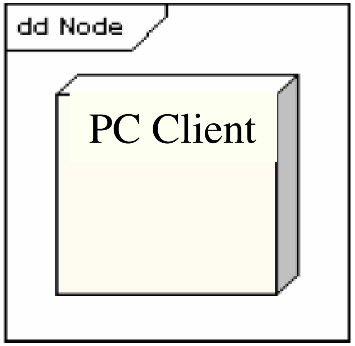
Deployment diagrams show the *hardware* for your system, the *software* that is installed on that hardware, and the *middleware* used to connect the disparate *machines* to one another!

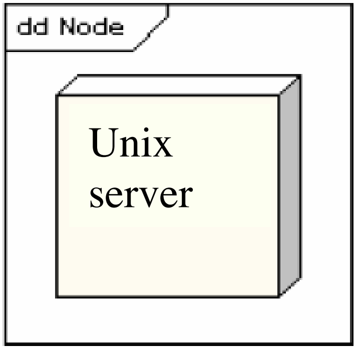
Visualizes the distribution of components in an application, it shows the configuration of the *hardware* elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

Node

A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.



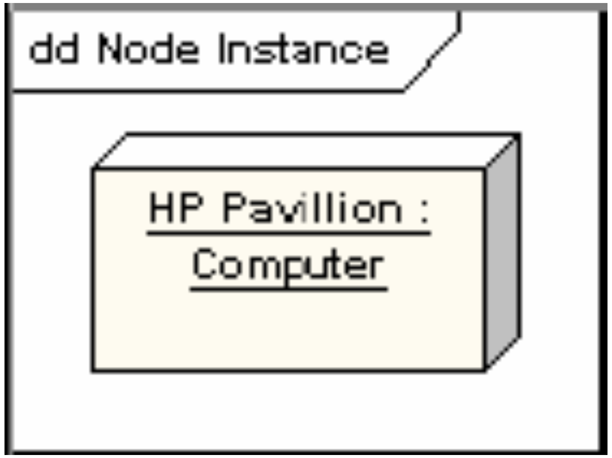




Node Instance

An **instance** can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon.

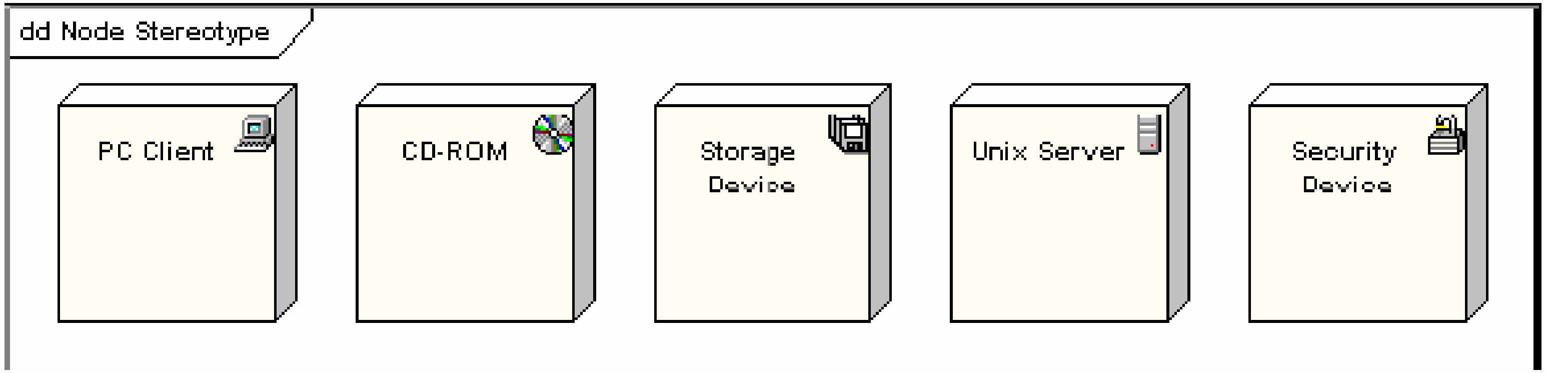
The following diagram shows a named instance of a computer



Node Sterotypes

In UML, a number of standard **stereotypes** are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc».

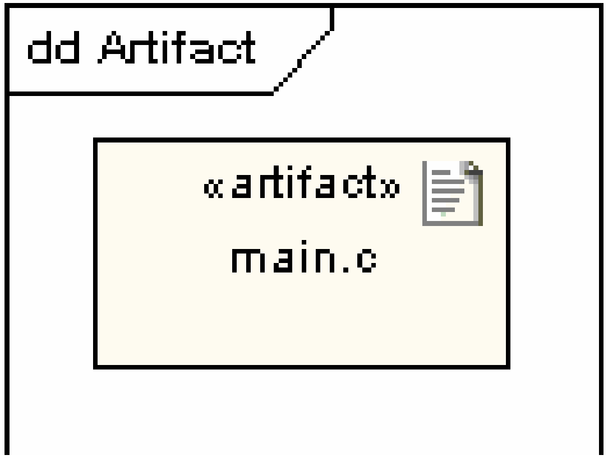
These will display an appropriate icon in the top right corner of the node symbol



Artifact

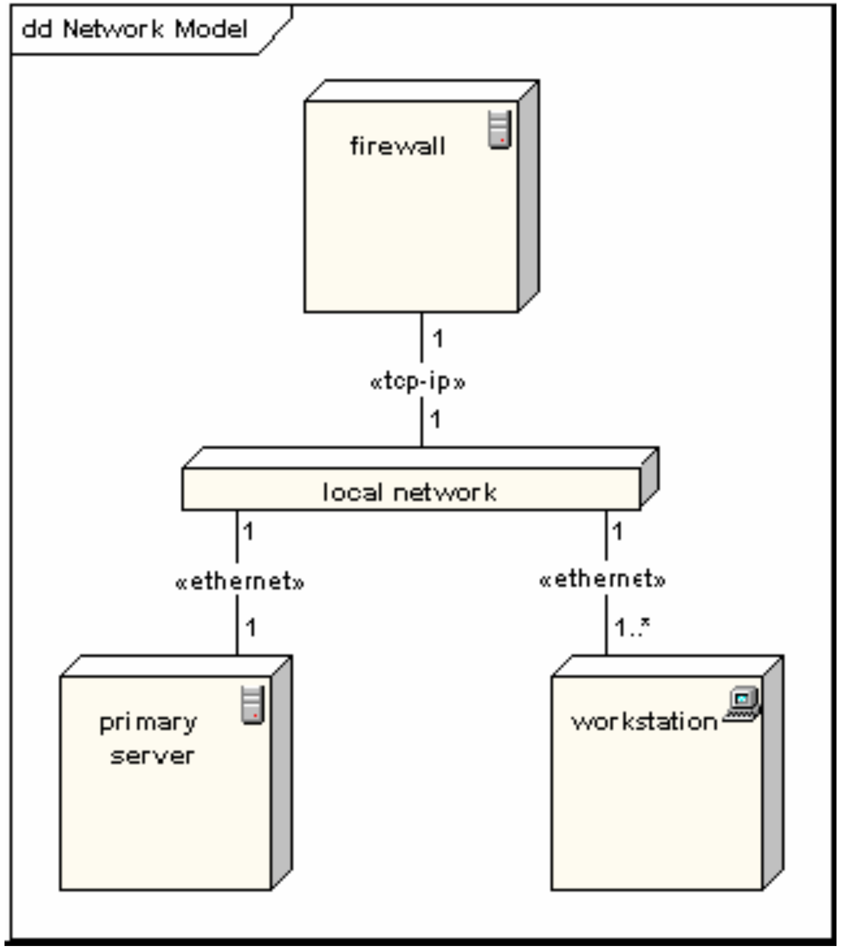
An **artifact** is a product of the software development process. That may include process models (e.g. use case models, design models etc), source files, executable files, design documents, test reports, prototypes, user manuals, etc.

An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown.



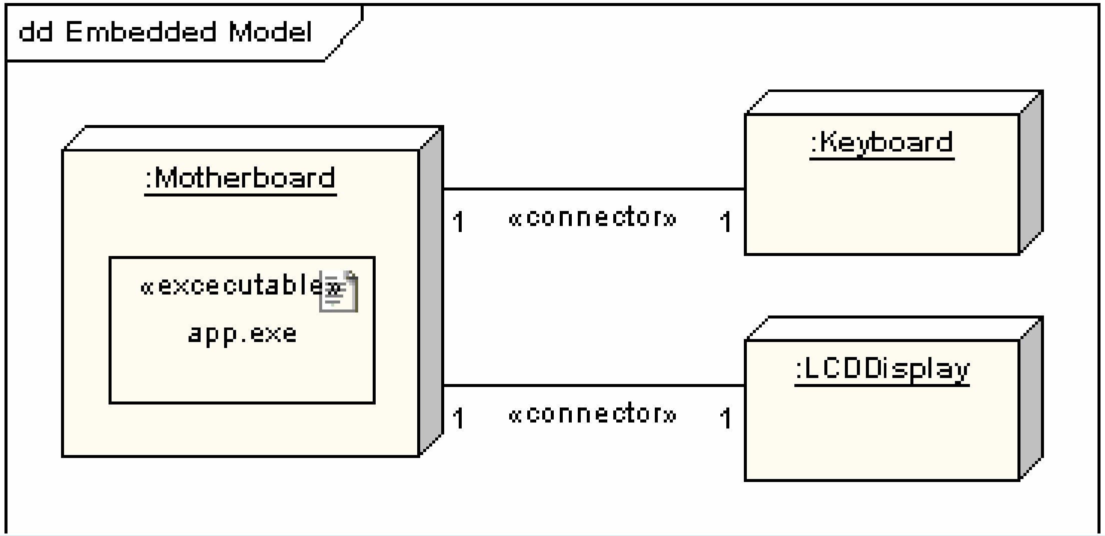
Association

In deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.



Node as Container

A node can contain other elements, such as components or artifacts.
The following diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.



Architectural Style vs Architecture

Architectural Style:

A pattern for a system layout

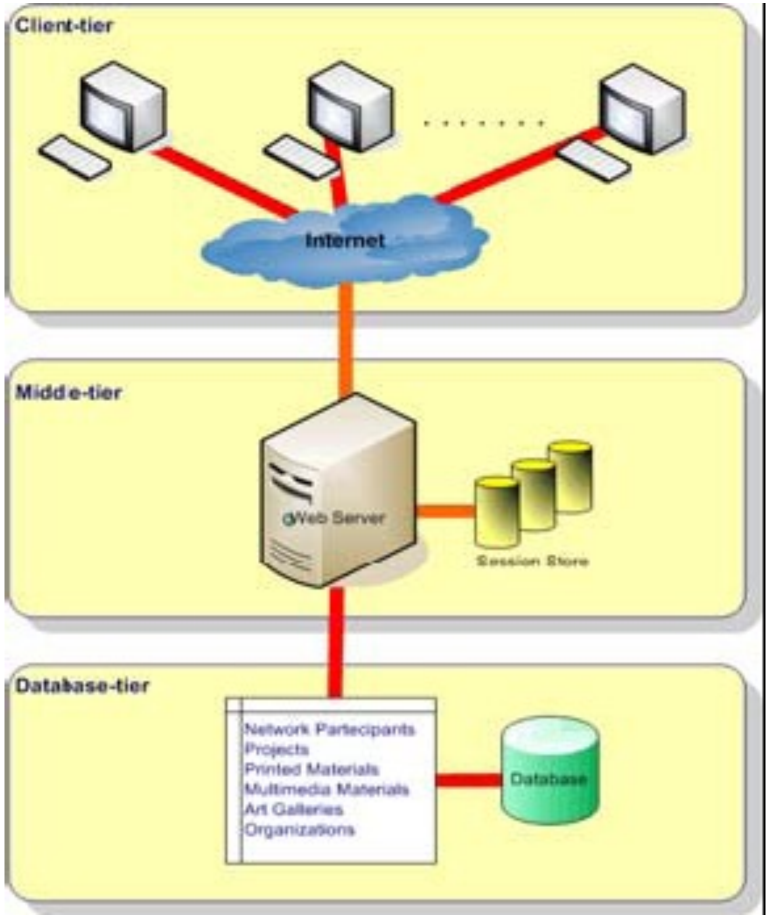
Software Architecture:

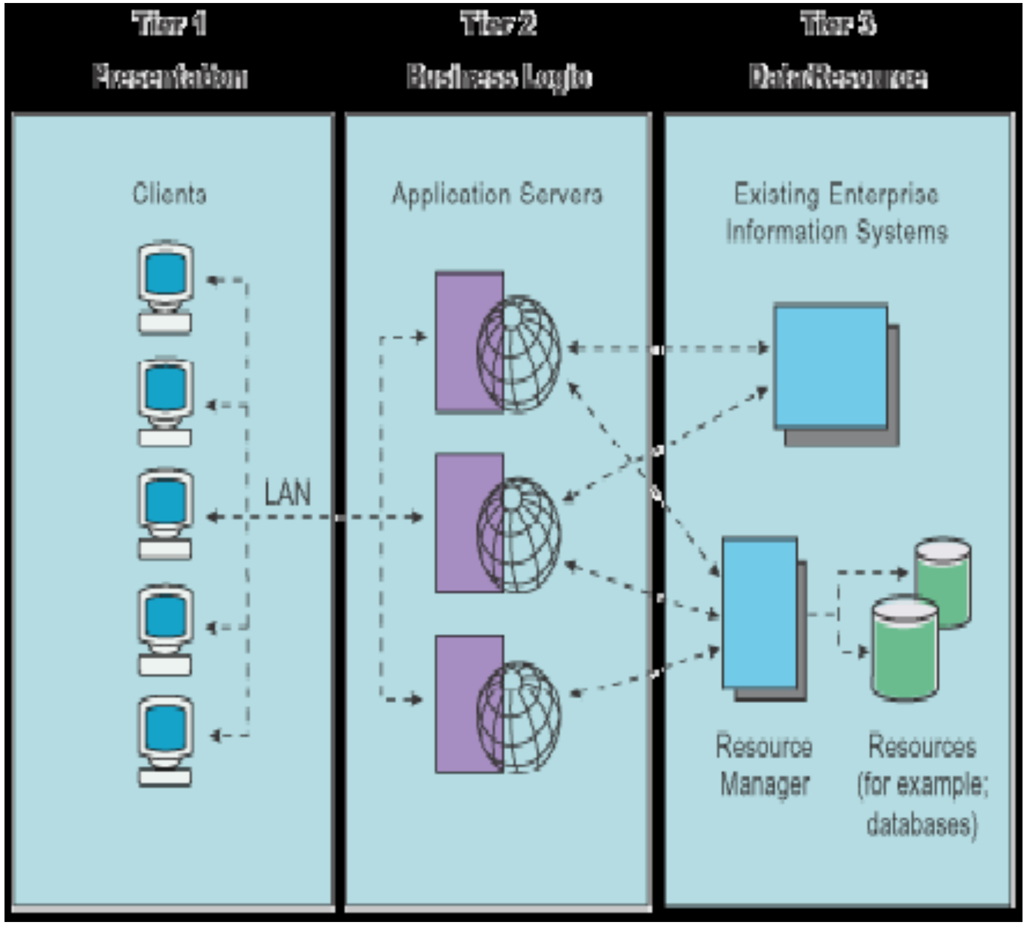
Instance of an architectural style.

Examples of Architectural Styles

- Layered Architectural style
  - Service-Oriented Architecture (SOA)
- Client/Server
- Peer-To-Peer
- Three-tier, Four-tier Architecture
- Repository
- Model-View-Controller
- Pipes and Filters

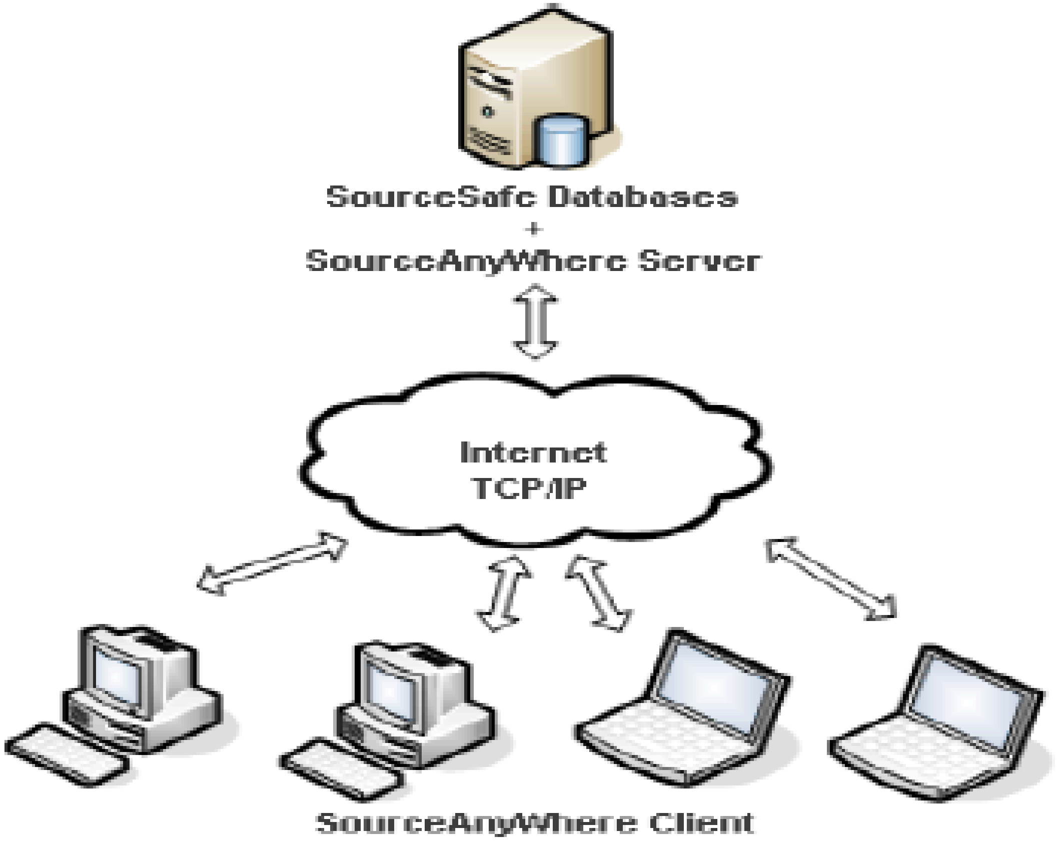
Example of three-tiers architectures





Many of real life web applications have three tier architectures

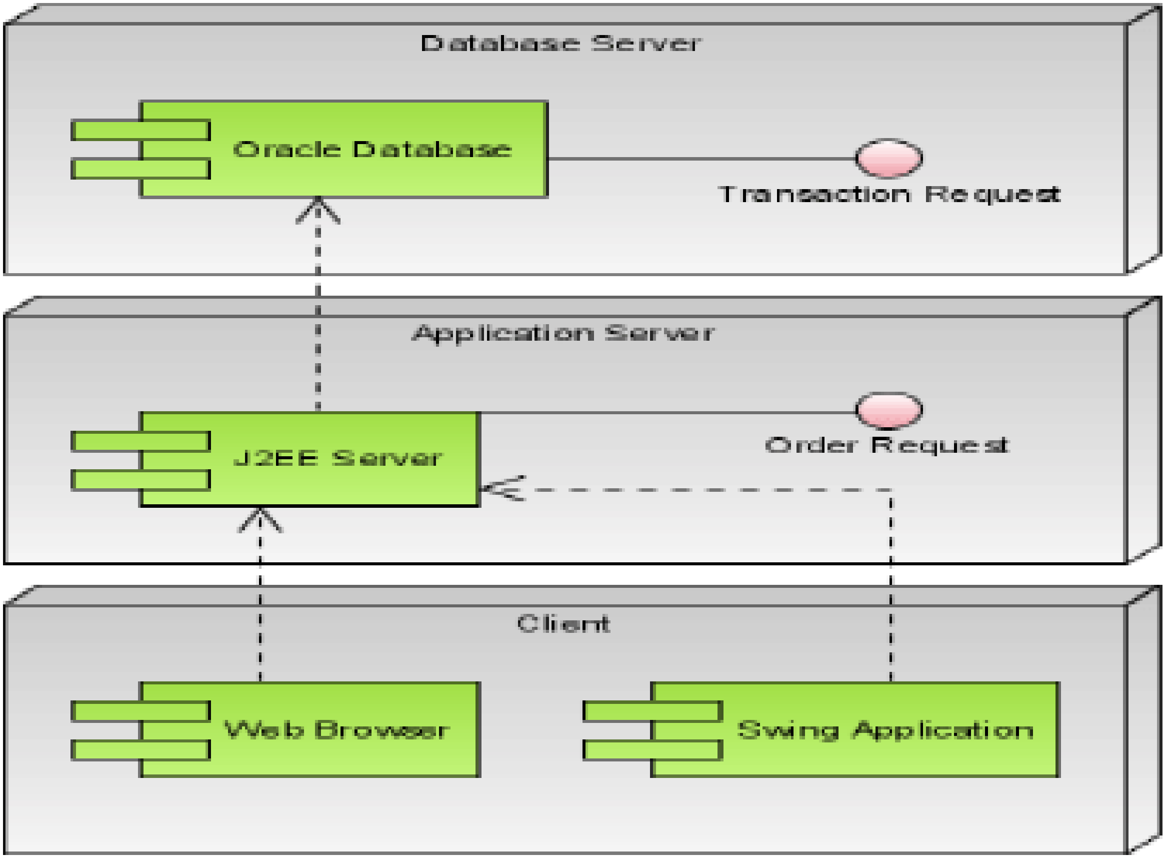
Example: Client server architectures



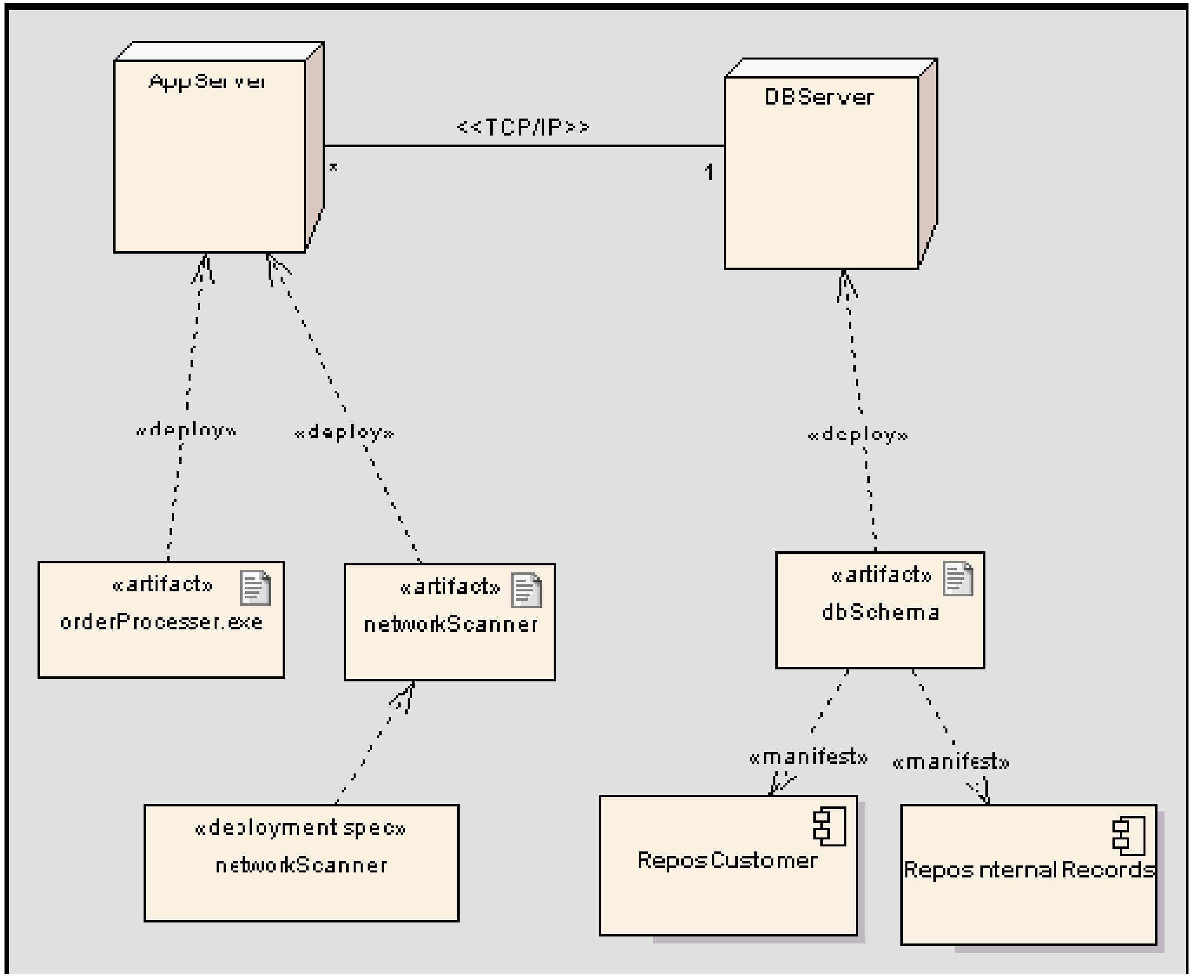
Deployment diagram for three tiers

UML 1.4

Components deployed in an architecture



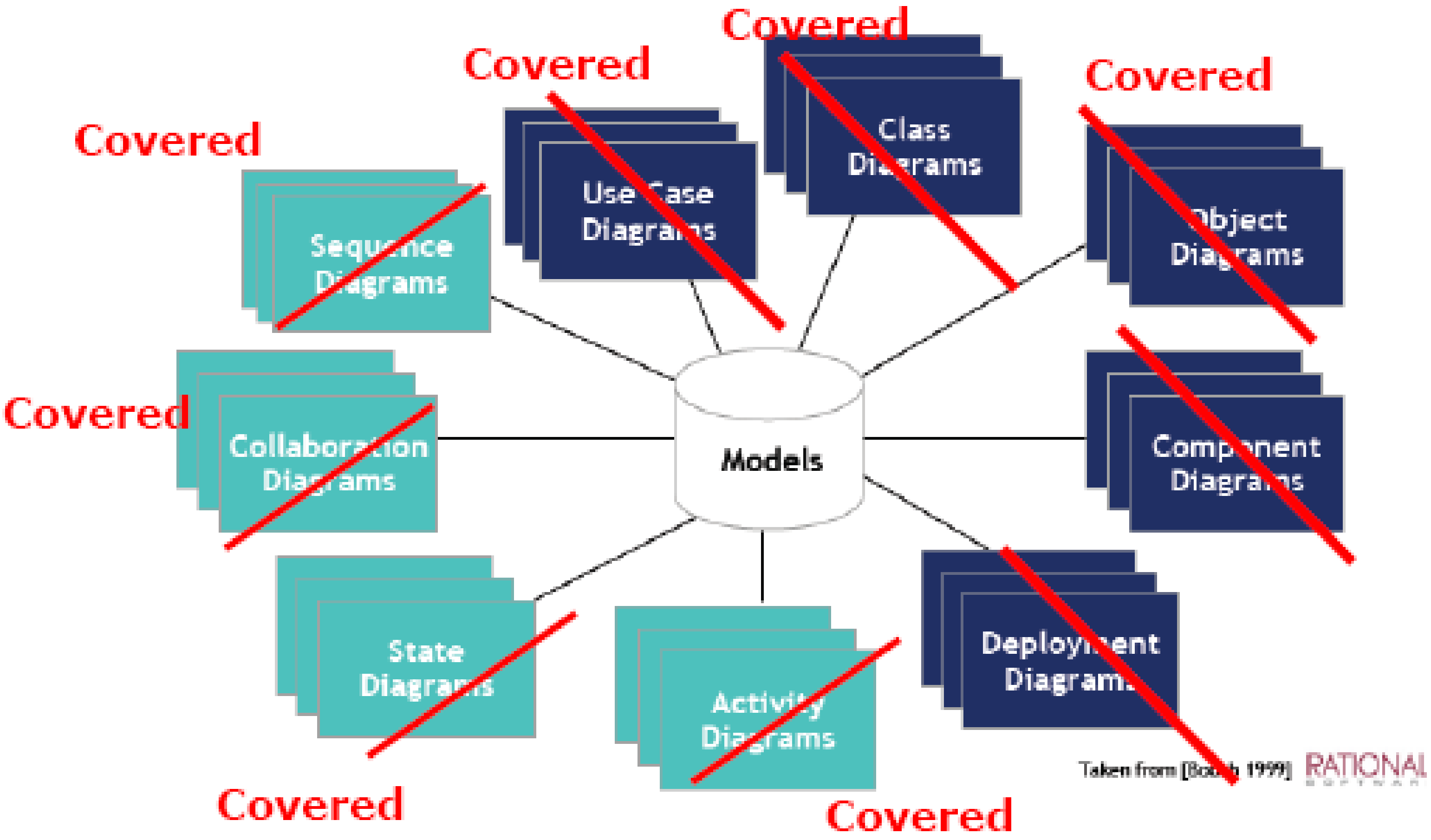
Example: Deployment Diagram for client server architectures



Exercise.

Depict a deployment diagram for an ATM machine.

Covered!?



Taken from [Booch 1999]

Key points

A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behaviour.

Context models show how a system that is being modeled is positioned in an environment with other systems and processes.

Structural models show the organization and architecture of a system. Use cases describe interactions between a system and external actors. Class diagrams are used to define the static structure of classes in a system and their associations using both data-driven and executable view points.

Behavioural models show how how system elements interactions. Use case diagrams, activity diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed taking the business view points or needs. Activity diagrams show how a business achieve its business process through interactions between use cases. Sequence diagrams show how a system achieve use cases through interactions between system objects.