Analysis, Design, and Software Architecture (BDSA)

*Paolo Tell*

# Introduction to UML

IT University
of Copenhagen

# What did we see so far … I

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software engineering is a collection of techniques, methodologies, and tools that help with the production of a high quality software system developed with a given budget before a given deadline while change occurs.
- Requirements engineering is the process of developing a software specification.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.

# What did we see so far … II

- Functional requirements statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.

- Non-functional requirements constrain the system being developed and the development process being used. Often relate to the emergent properties of the system and therefore apply to the system as a whole—they concern the architecture.

- Domain requirements are constraints on the system from the domain of operation—the application domain.

- The software requirements document is an agreed statement of the system requirements that should be organized so that both system customers and software developers can use it.

- The requirements engineering process an iterative process that includes requirements elicitation, specification, and validation.

- Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.

# What did we see so far … III

- Requirement elicitation concepts
  - Completeness, Consistency, Clarity, and Correctness
  - Realism, Verifiability, and Traceability

- Requirements elicitation activities
  - Identify Actors, Scenarios, and Use Cases
  - Identify Relationships between Actors and Use Cases
  - Identify Initial Analysis Objects
  - Identify Non-functional Requirements

- Documenting requirements elicitation
  - Requirements analysis document (RAD)

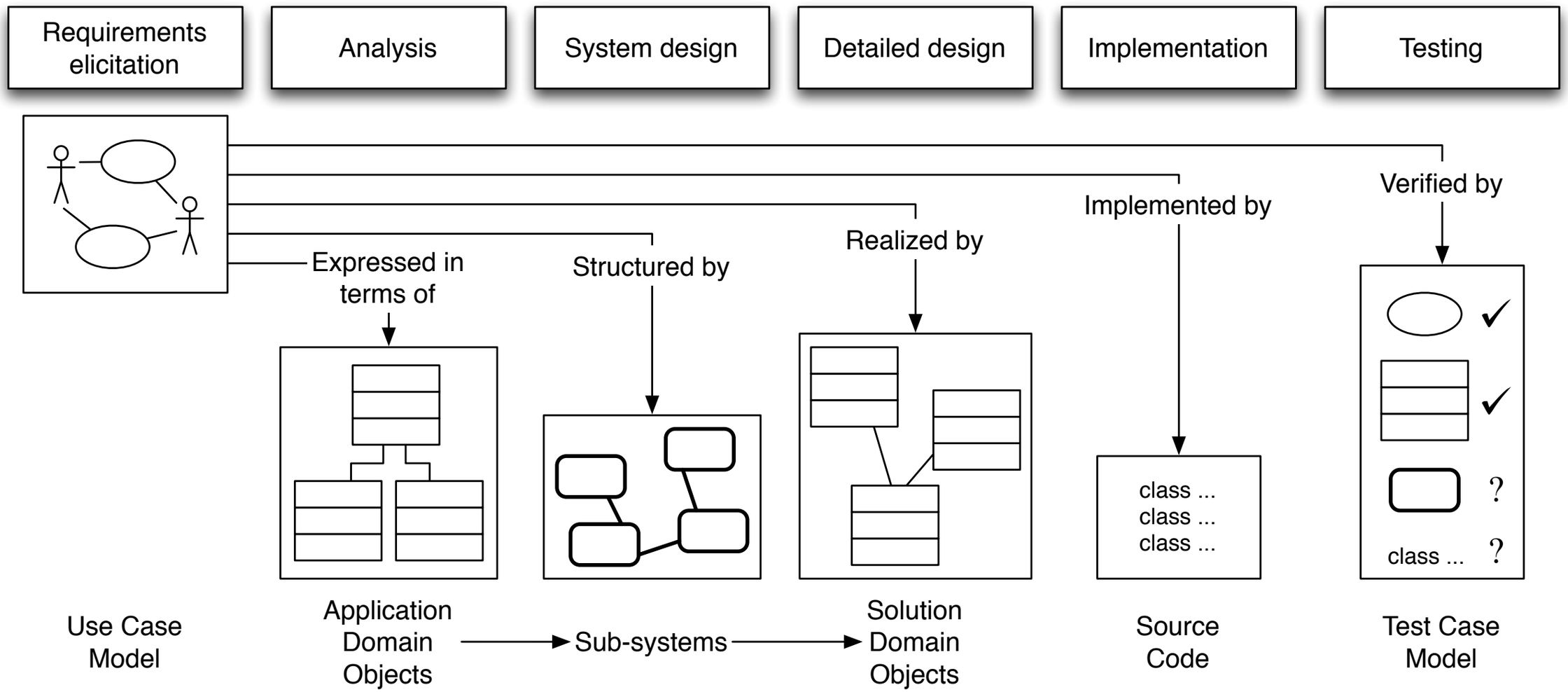


**Requirements Analysis Document**

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Scope of the system
   - 1.3 Objectives and success criteria of the project
   - 1.4 Definitions, acronyms, and abbreviations
   - 1.5 References
   - 1.6 Overview
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
     - 3.3.1 Usability
     - 3.3.2 Reliability
     - 3.3.3 Performance
     - 3.3.4 Supportability
     - 3.3.5 Implementation
     - 3.3.6 Interface
     - 3.3.7 Packaging
     - 3.3.8 Legal
   - 3.4 System models
     - 3.4.1 Scenarios
     - 3.4.2 Use case model
     - *3.4.3 Object model*
     - *3.4.4 Dynamic model*
     - 3.4.5 User interface—navigational paths and screen mock-ups
4. Glossary

**Figure 4-16** Outline of the Requirements Analysis Document (RAD). during analysis (see next chapter).

# What did we see so far … IV




- Software lifecycle activities

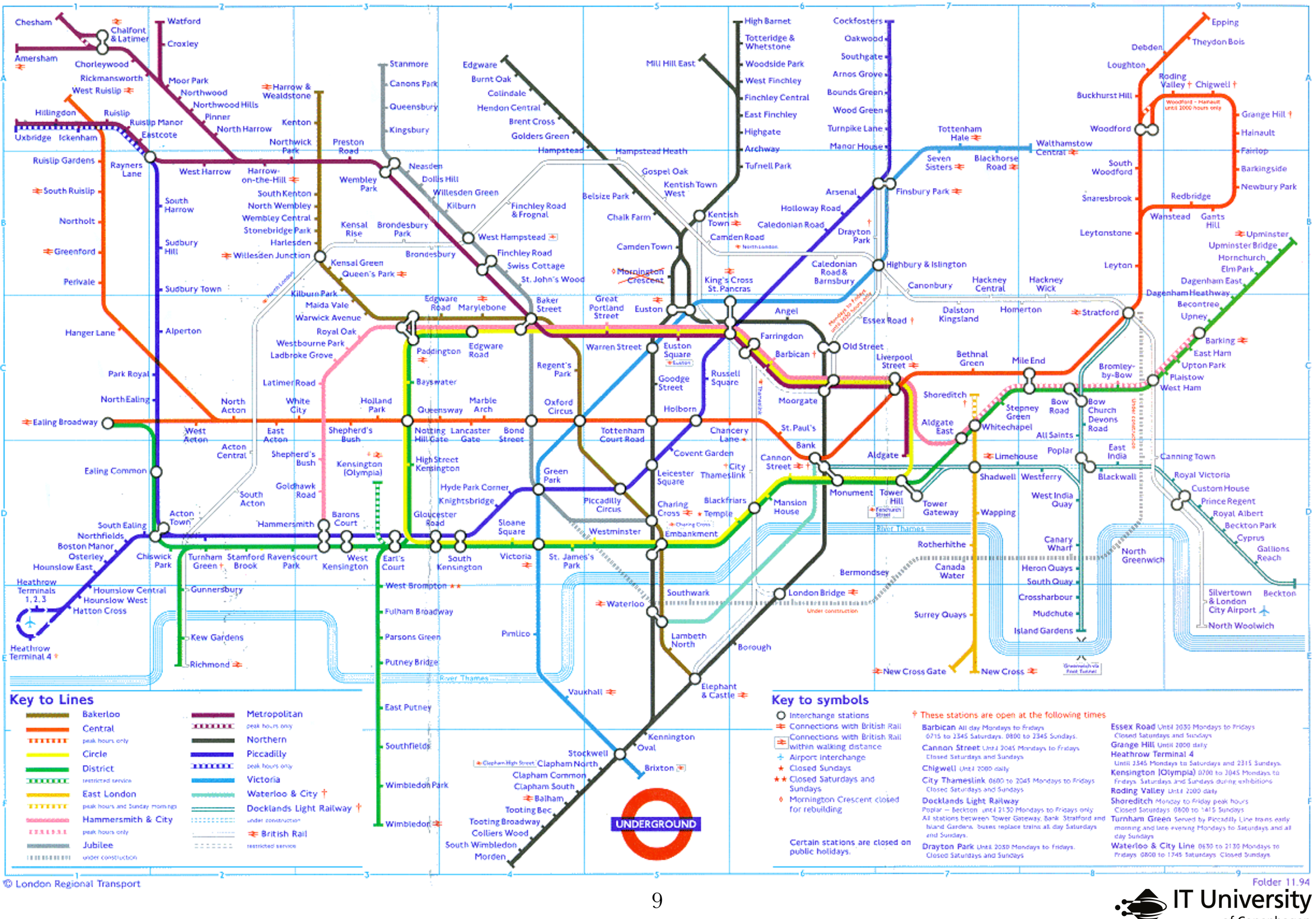
# Outline

- Literature
  - [OOSE] ch. 2
  - (Optional) [SE10] ch. 5

- Topics covered:
  - Modeling
  - UML and its history
  - UML diagrams
    - Brief description
    - Example(s)
    - Comment (or another example)
    - Template from [uml-diagrams.org]
  - A deeper dive into class diagrams

# Modeling


GREEN PARK
Lift
to Piccadilly line
1 STRATFORD
2 NORTH GREENWICH
fly to hungary & poland
london luton
budapest
warsaw • gdansk
katowice/pyrzowice



# Modeling concepts

- System
  - an organised set of communicating parts.
  - a systems is composed of a set of sub-systems.
  - models are used to handle complex systems.
    - often several models are used to describe a system.

- Model
  - an abstraction describing a system or sub-system (divide-et-impera).

- Modeling
  - Modelling means constructing an abstraction of a system that focuses on interesting aspects and ignores irrelevant details.
  - A rule of thumb is that each entity should contain at most 7 ± 2 parts [Miller, 1956].

- View
  - depicts selected aspects of a model.

- Notation
  - is a set of graphical or textual rules for depicting models and views.

# Systems, Models, and Views

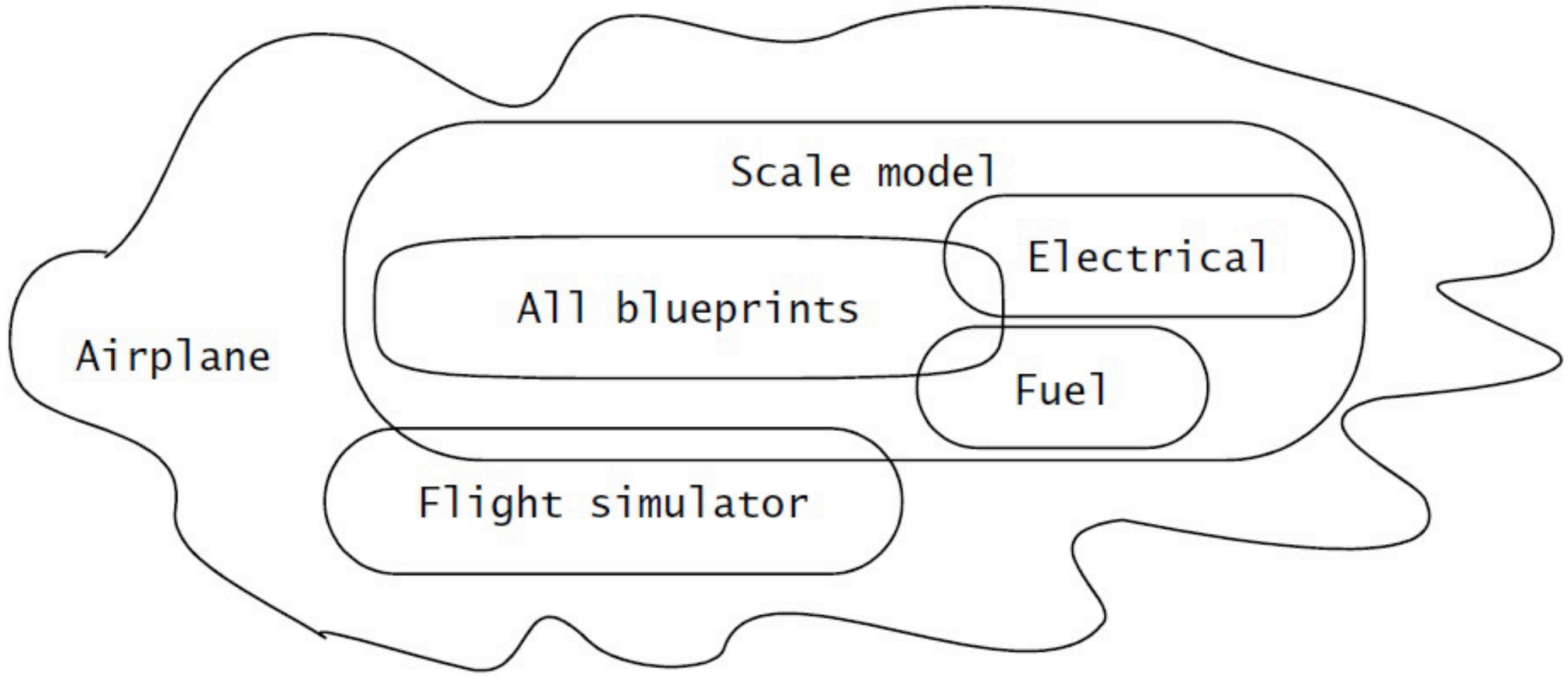



**Figure 2-6** A model is an abstraction describing a subset of a system. A view depicts selected aspects of a model. Views and models of a single system may overlap each other.

# Object-oriented modelling

- Application Domain (Analysis)
  - The environment in which the system is operating.
  - The user's problem.
  - In OO, we model this domain through object-oriented analysis (OOA).

- Solution Domain (Design, Implementation)
  - The technologies used to build the system.
  - The software engineer's problem.
  - In OO, we model this domain through object-oriented design (OOD).

- Both domains contain abstractions that we can use for the construction of the system model.


# The Unified Modeling Language (UML) and its history

# Unified Modeling Language (UML)

- Object Management Group (OMG) definition
  - "The Unified Modelling Language is a visual language for specifying, constructing, and documenting the artifacts of systems" [OMG].
- Adopted in 1997 as a standard by OMG.
- Published in 2000 by the International Organization for Standardisation (ISO) as an approved ISO standard.

# The 3 amigos

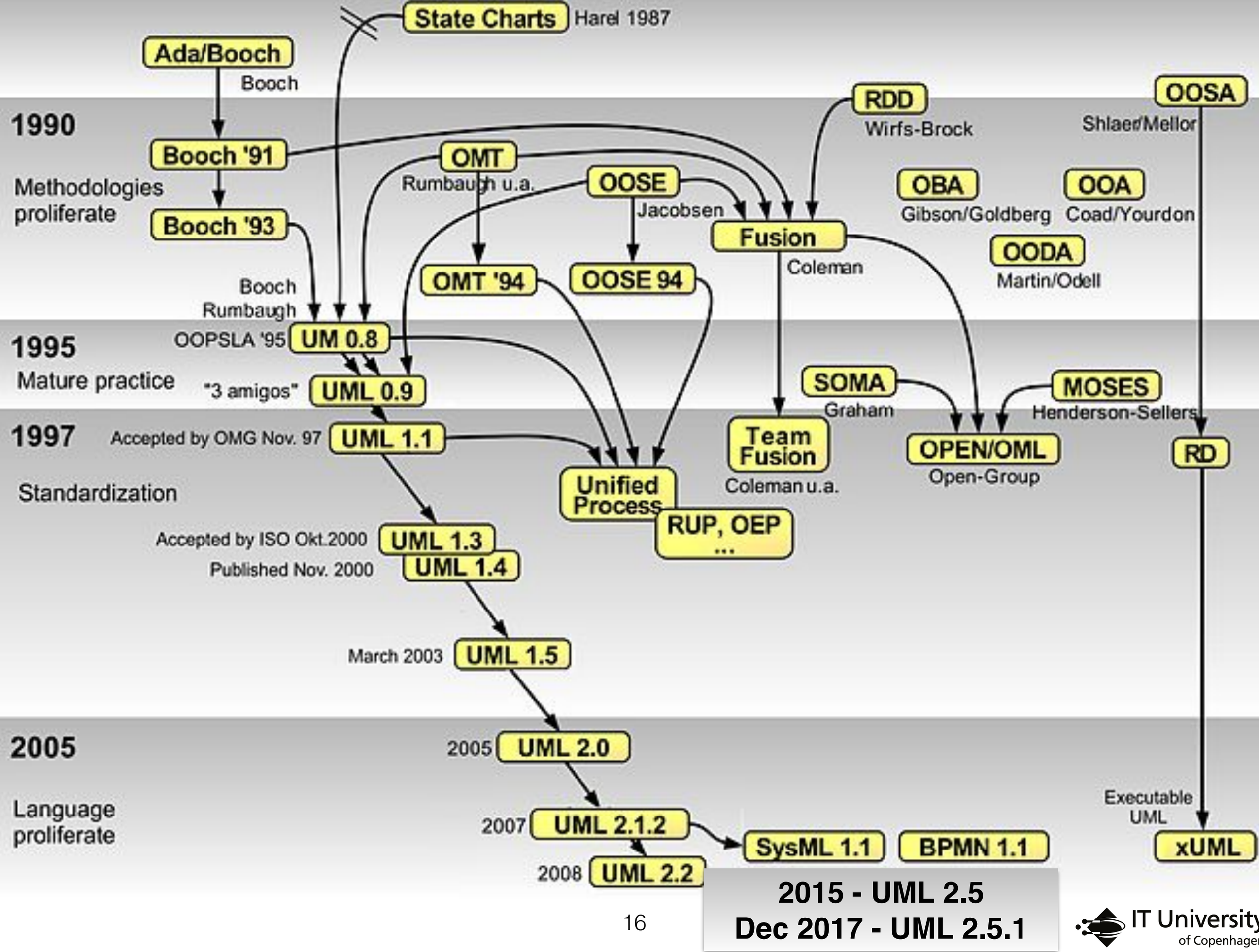
25 year at General Electric Research, where he developed OMT, joined (IBM) Rational in 1994, CASE tool OMTool

At Ericsson until 1994, developed use cases and the CASE tool Objectory, at IBM Rational since 1995

Developed the Booch method ("clouds"), ACM Fellow 1995, and IBM Fellow 2003

- Convergence of different notations used in object-oriented methods, mainly
  - OMT (James Rumbaugh and colleagues),
  - OOSE (Ivar Jacobson),
  - Booch (Grady Booch)

- They also developed the Rational Unified Process (RUP), which became the Unified Process in 1999: a rich iterative process


State Charts
Harel 1987
Ada/Booch
Booch
1990
Booch '91
Booch '93
Methodologies
proliferate
OMT
Rumbaugh u.a.
OOSE
Jacobsen
RDD
Wirfs-Brock
OOSA
Shlaer/Mellor
OBA
Gibson/Goldberg
OOA
Coad/Yourdon
Fusion
Coleman
OODA
Martin/Odell
Booch
Rumbaugh
OOPSLA '95
UM 0.8
OMT '94
OOSE 94
1995
Mature practice
"3 amigos"
UML 0.9
SOMA
Graham
MOSES
Henderson-Sellers
1997
Accepted by OMG Nov. 97
UML 1.1
Team
Fusion
Coleman u.a.
OPEN/OML
Open-Group
RD
Unified
Process
RUP, OEP
...
Standardization
Accepted by ISO Okt.2000
UML 1.3
Published Nov. 2000
UML 1.4
March 2003
UML 1.5
2005
2005
UML 2.0
Language
proliferate
2007
UML 2.1.2
SysML 1.1
BPMN 1.1
Executable
UML
xUML
2008
UML 2.2
2015 - UML 2.5
Dec 2017 - UML 2.5.1

# Why UML?

- Provide users an expressive visual modelling language.
- Facilitate communication among developers. UML became the lingua franca in modelling.
- Incorporate the object-oriented community's consensus on core modelling concepts.
- Provide extensibility and specialisation mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modelling language.
- Support the creation of object-oriented CASE tools.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- Win the 'war of notations'.

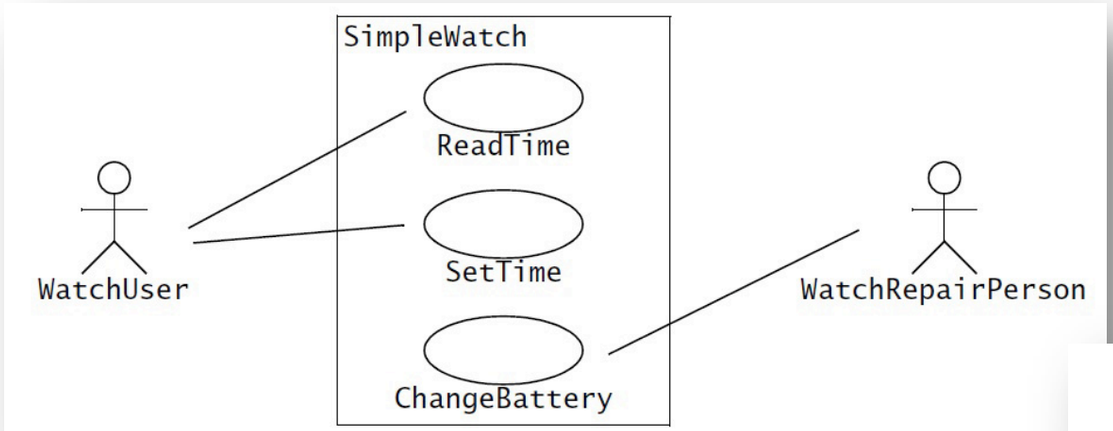
# UML

- 3 views on UML
  - UML as sketch.
  - UML as blueprint.
    - Before ~ specification.
    - After ~ documentation.
  - UML as a programming language.

- In the OOSE book
  - "Just" a diagramming notation standard.
  - Trivial and relatively unimportant.
  - Not a method, process, or design guide.

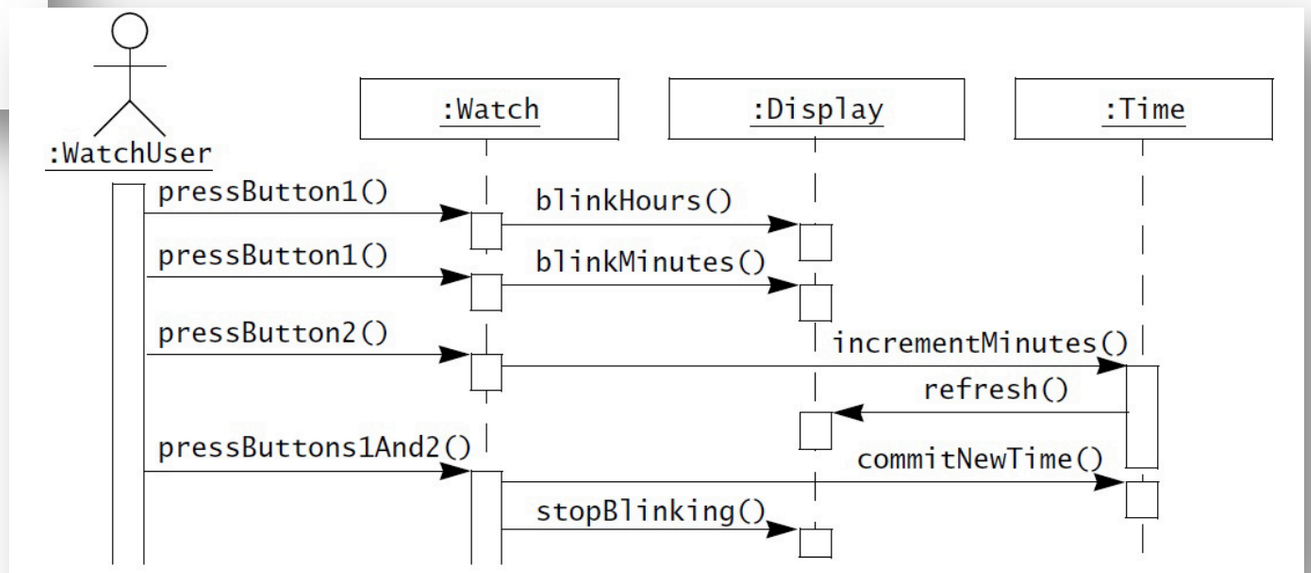
- In this course
  - You need to be able to draw syntactically correct UML diagrams – important part of the exam.

# Examples

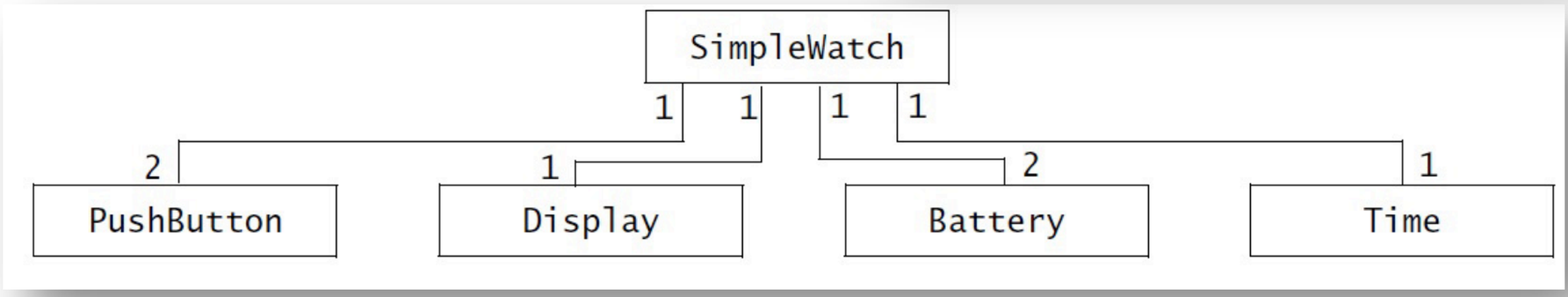


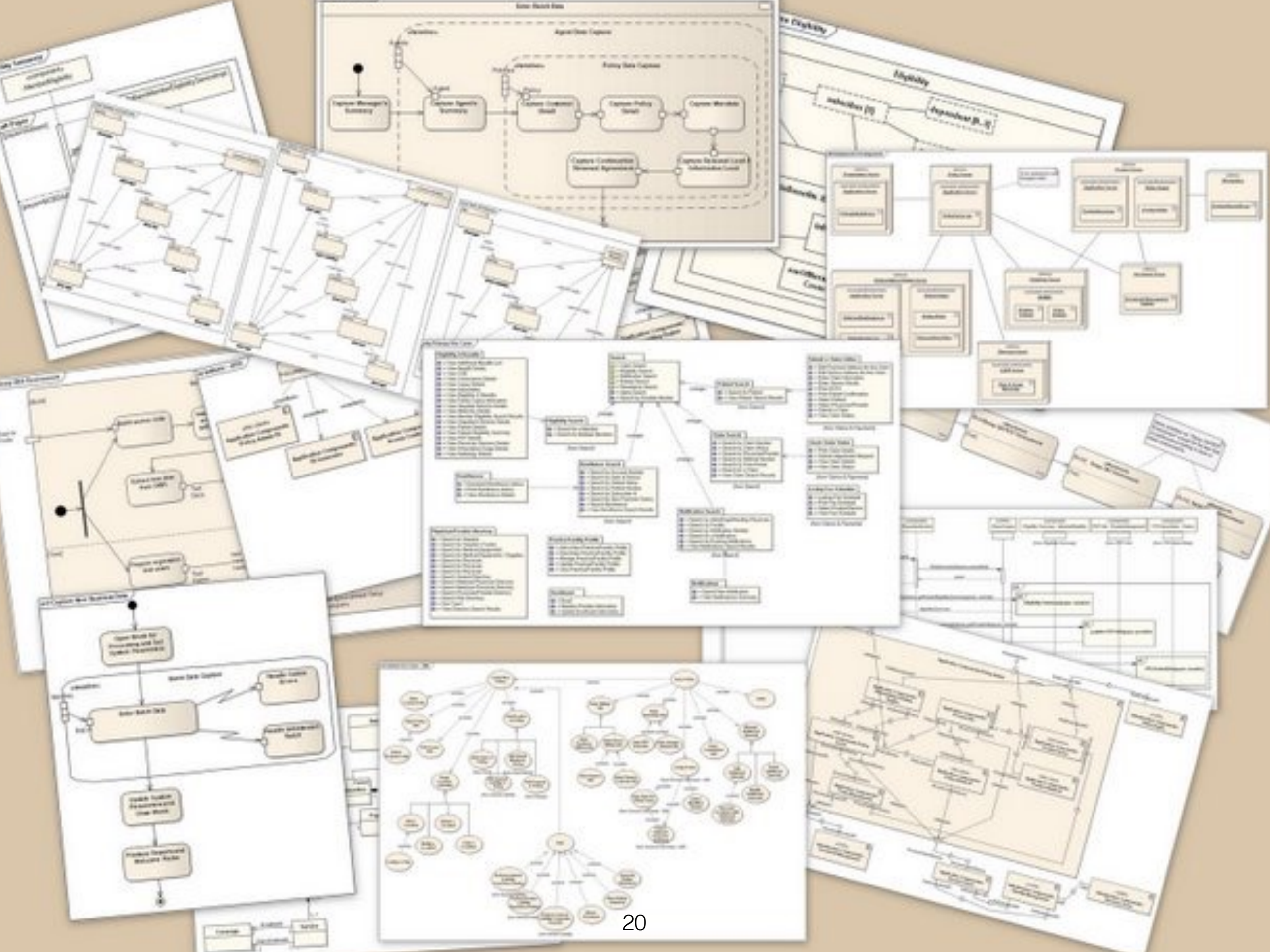

Use Case Diagram

Sequence Diagram
one of the Interaction Diagrams

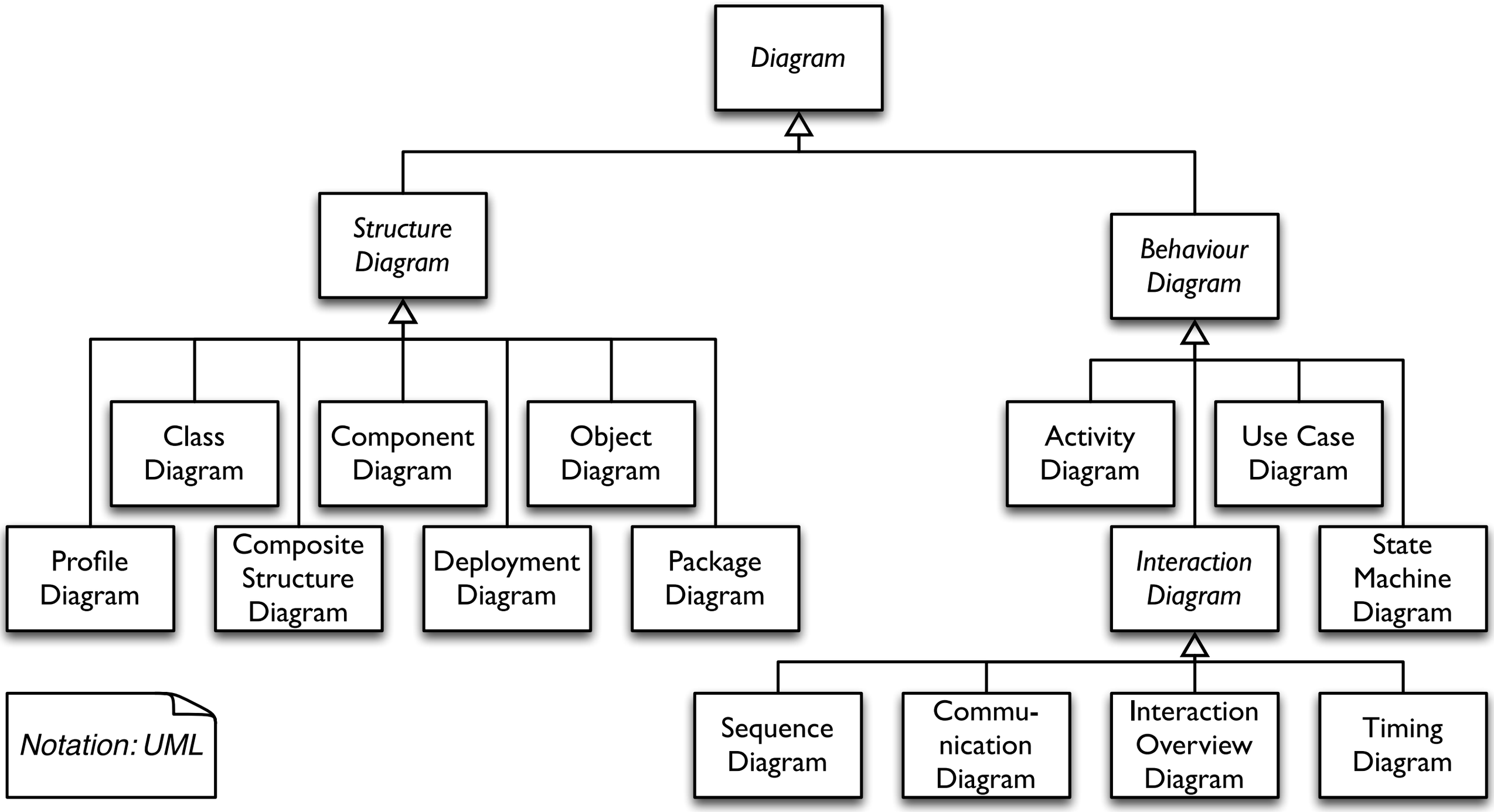


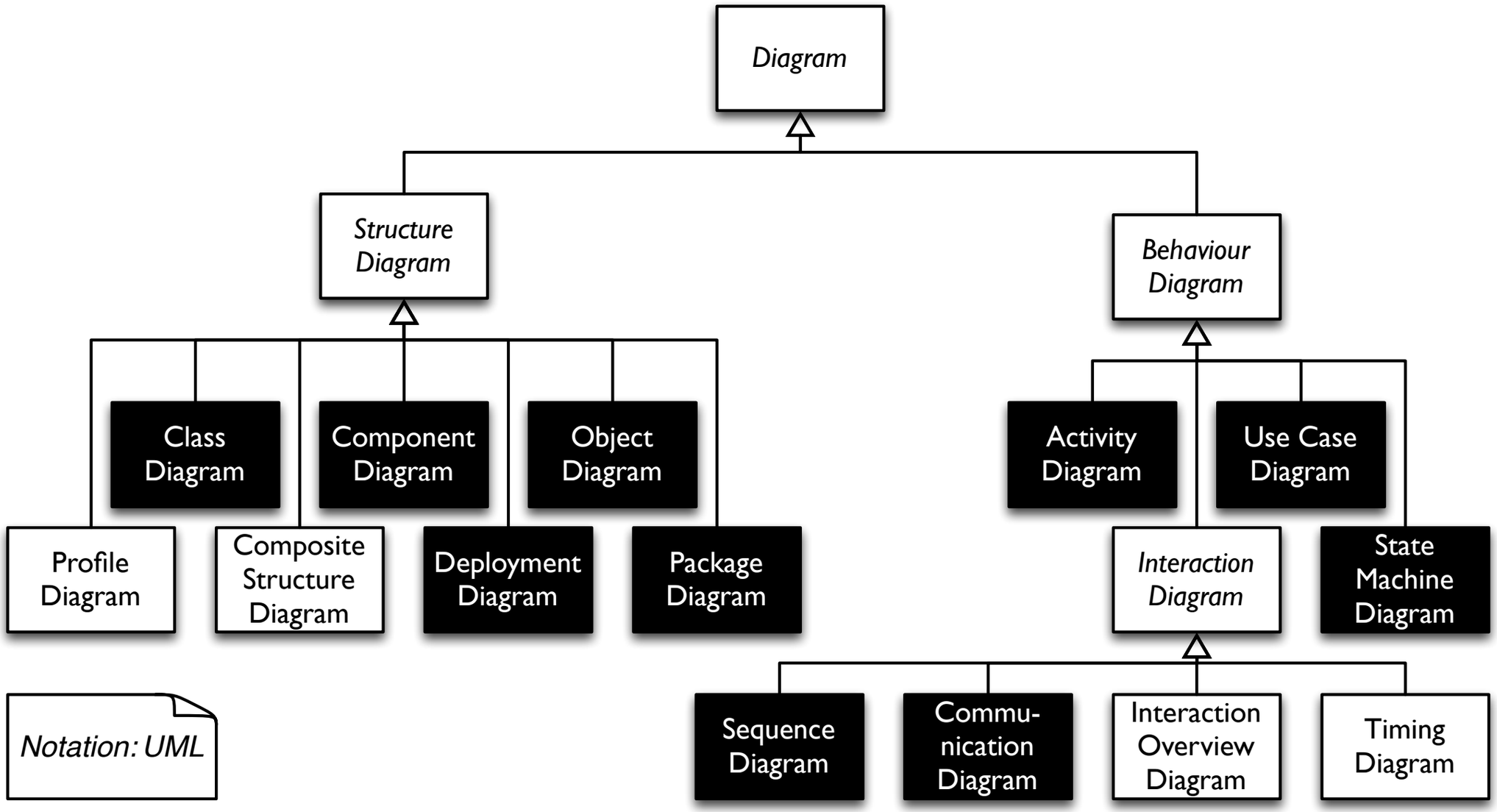

Object Diagram



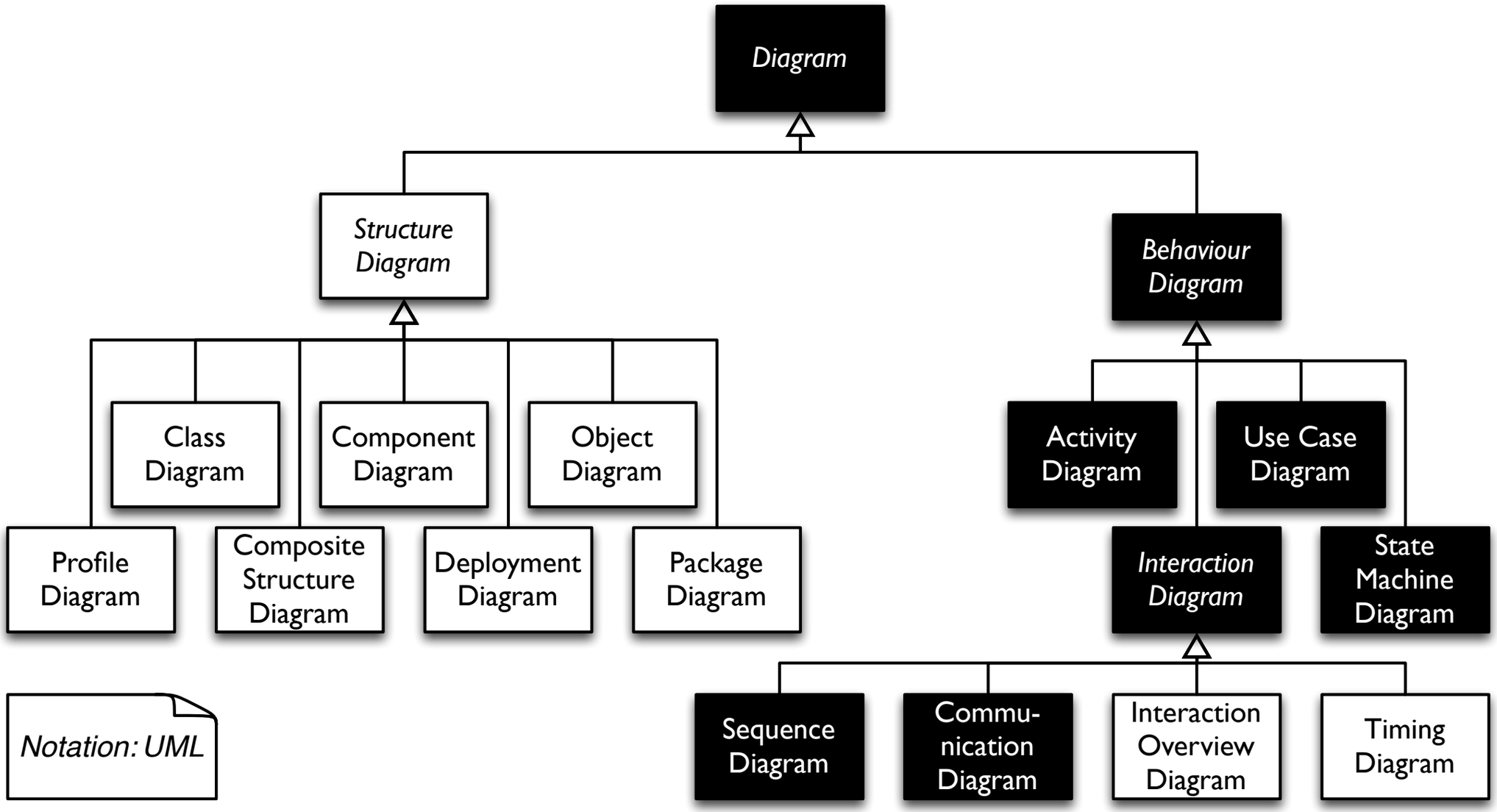


# Hierarchy of diagrams in UML 2.x



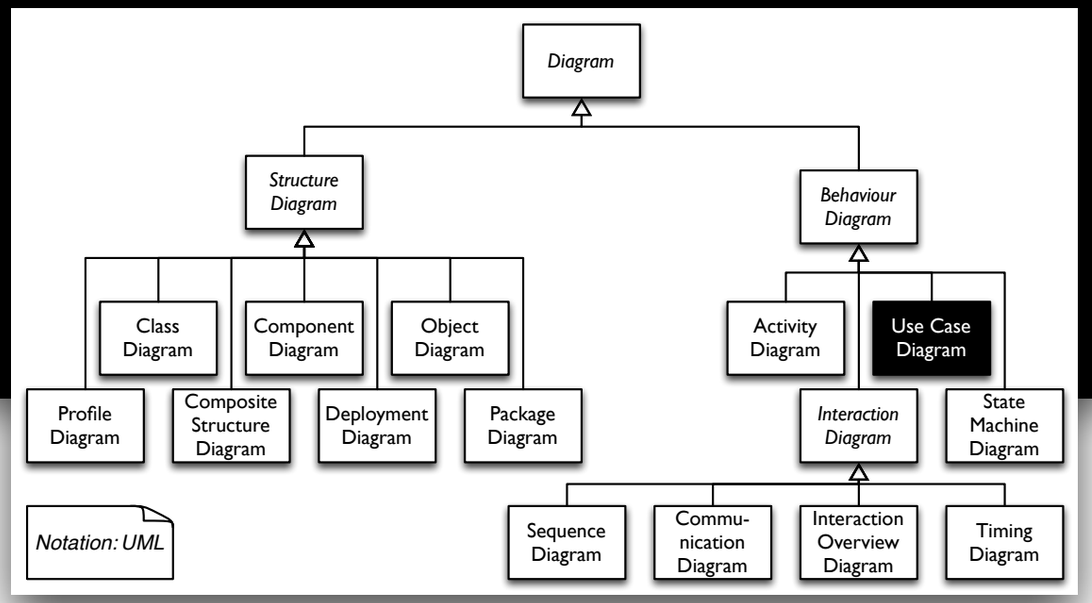



# Diving into UML

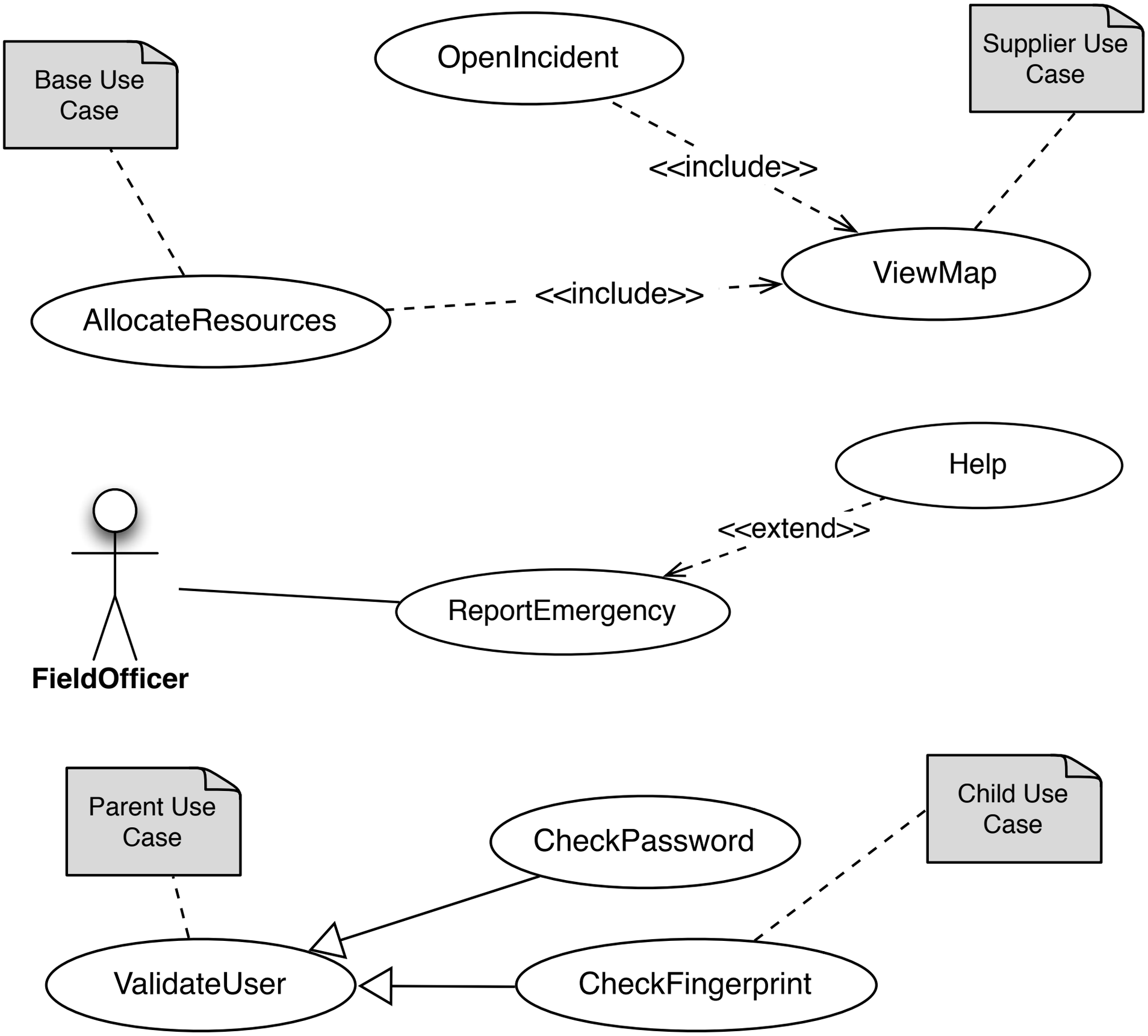


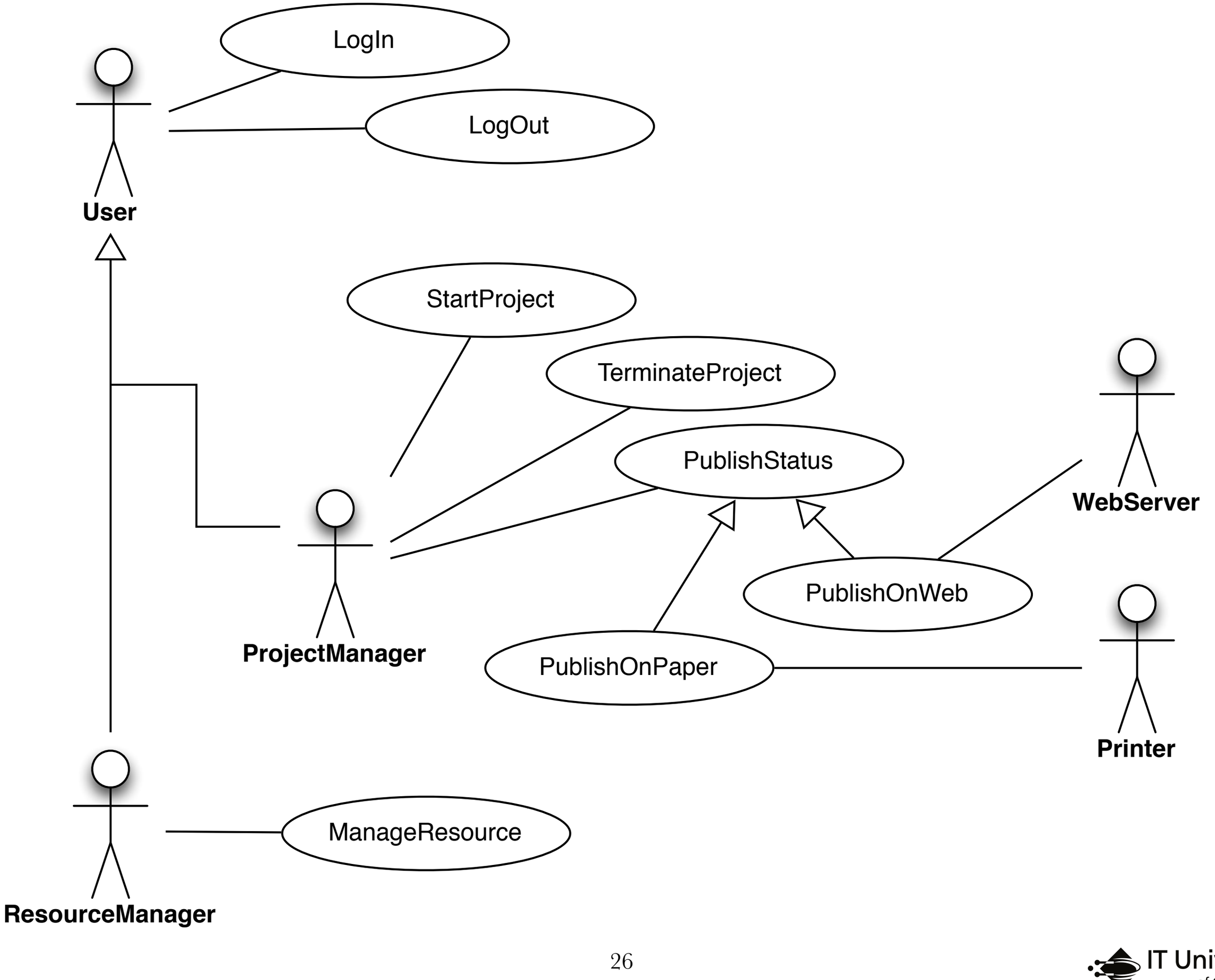

# Diving into UML (1st part)

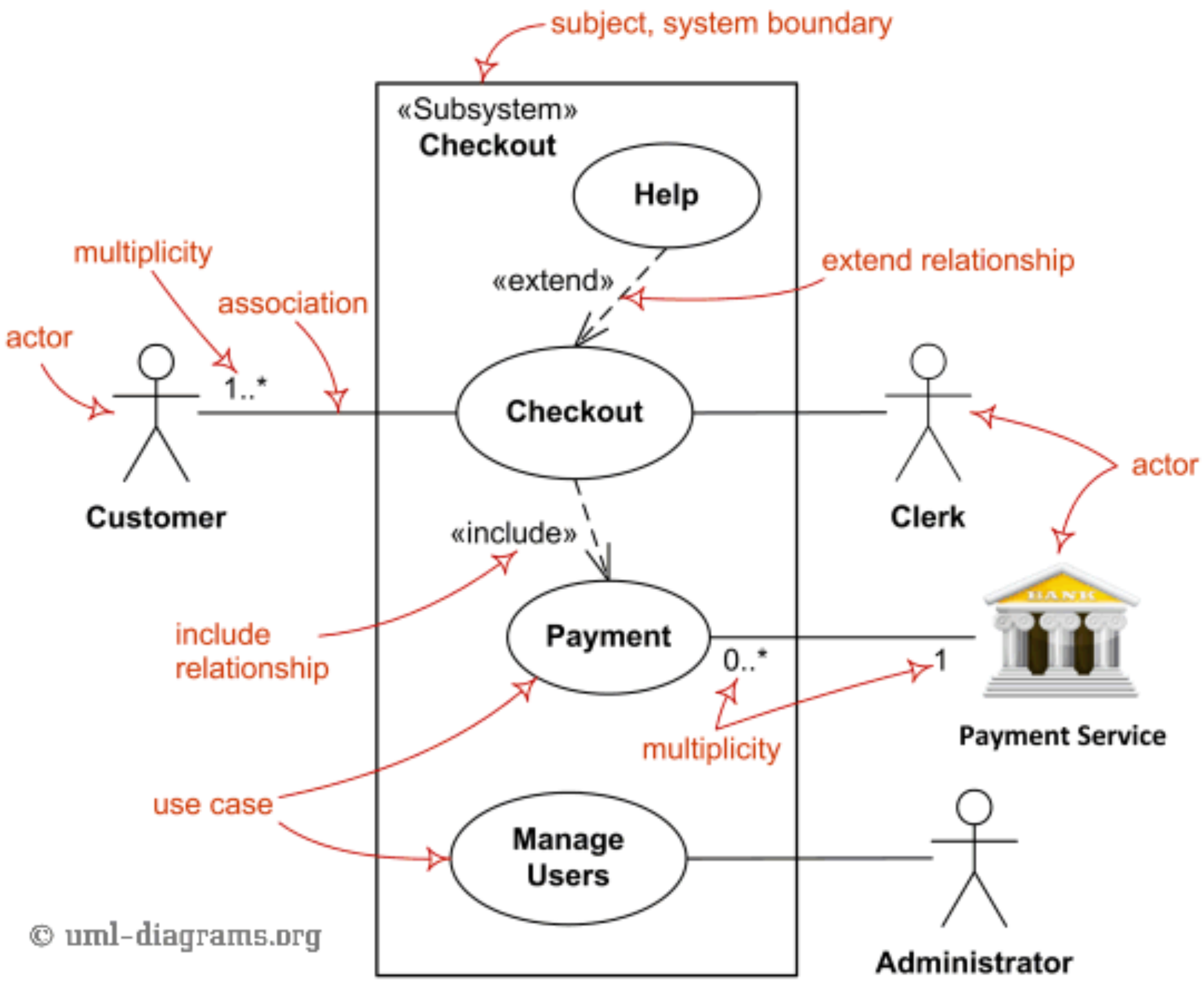
# Use case diagram




- Behaviour diagrams.
- They capture requirements.
- A use case captures a specific functionality offered by the system.
- They abstract scenarios.
- They comprise of two parts:
  - the use case diagram (the “sticky man” diagram).
    - actors (roles, not only humans), use cases, and associations (includes, extends, generalisation).
  - the use case text.


Base Use Case
OpenIncident
Supplier Use Case
<<include>>
ViewMap
AllocateResources
<<include>>
Help
<<extend>>
ReportEmergency
FieldOfficer
Parent Use Case
CheckPassword
Child Use Case
ValidateUser
CheckFingerprint


LogIn
LogOut
User
StartProject
TerminateProject
PublishStatus
WebServer
ProjectManager
PublishOnWeb
PublishOnPaper
Printer
ManageResource
ResourceManager


subject, system boundary
«Subsystem»
Checkout
Help
«extend»
extend relationship
multiplicity
association
actor
1..*
Checkout
Customer
Clerk
actor
«include»
include
relationship
Payment
0..*
1
multiplicity
Payment Service
use case
Manage
Users
Administrator
© uml-diagrams.org

| | |
|---|---|
| *Use case name* | `ReportEmergency` |
| *Participating actors* | Initiated by `FieldOfficer`<br>Communicates with `Dispatcher` |
| *Flow of events* | 1. The `FieldOfficer` activates the "Report Emergency" function of her terminal.<br>2. `FRIEND` responds by presenting a form to the officer. *The form includes an emergency type menu (general emergency, fire, transportation) and location, incident description, resource request, and hazardous material fields.*<br>3. The `FieldOfficer` completes the form by *specifying minimally the emergency type and description fields*. The `FieldOfficer` may also describe possible responses to the emergency situation *and request specific resources*. Once the form is completed, the `FieldOfficer` submits the form.<br>4. `FRIEND` receives the form and notifies the `Dispatcher` *by a pop-up dialog*.<br>5. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. *All the information contained in the `FieldOfficer's` form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.*<br>6. `FRIEND` displays the acknowledgment and the selected response to the `FieldOfficer`. |
| *Entry condition* | • ... |

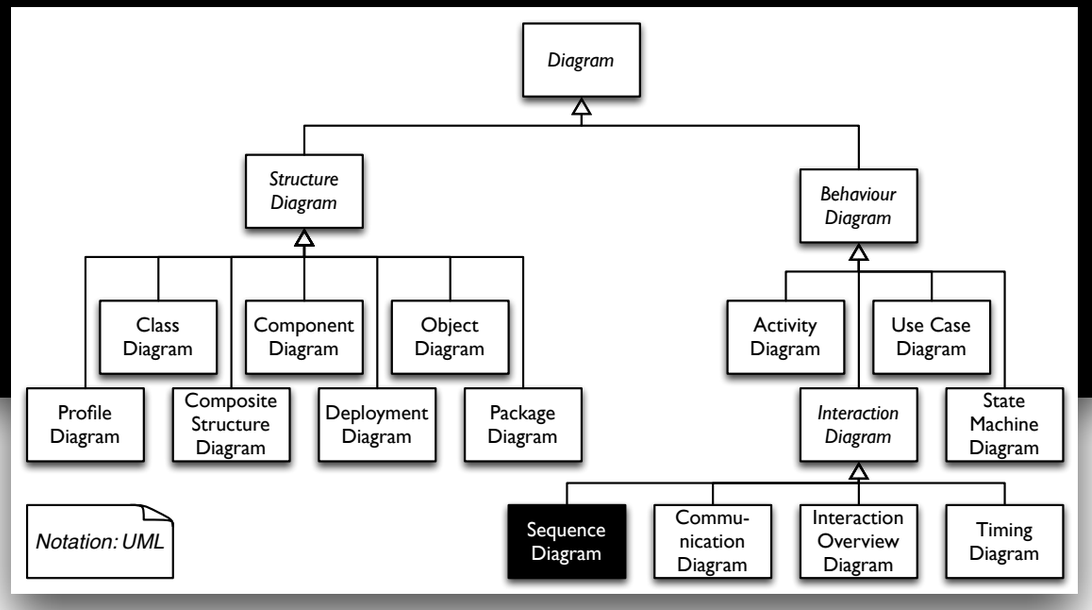
**Figure 4-10** Refined description for the `ReportEmergency` use case. Additions emphasized in *italics*.

# Beware!

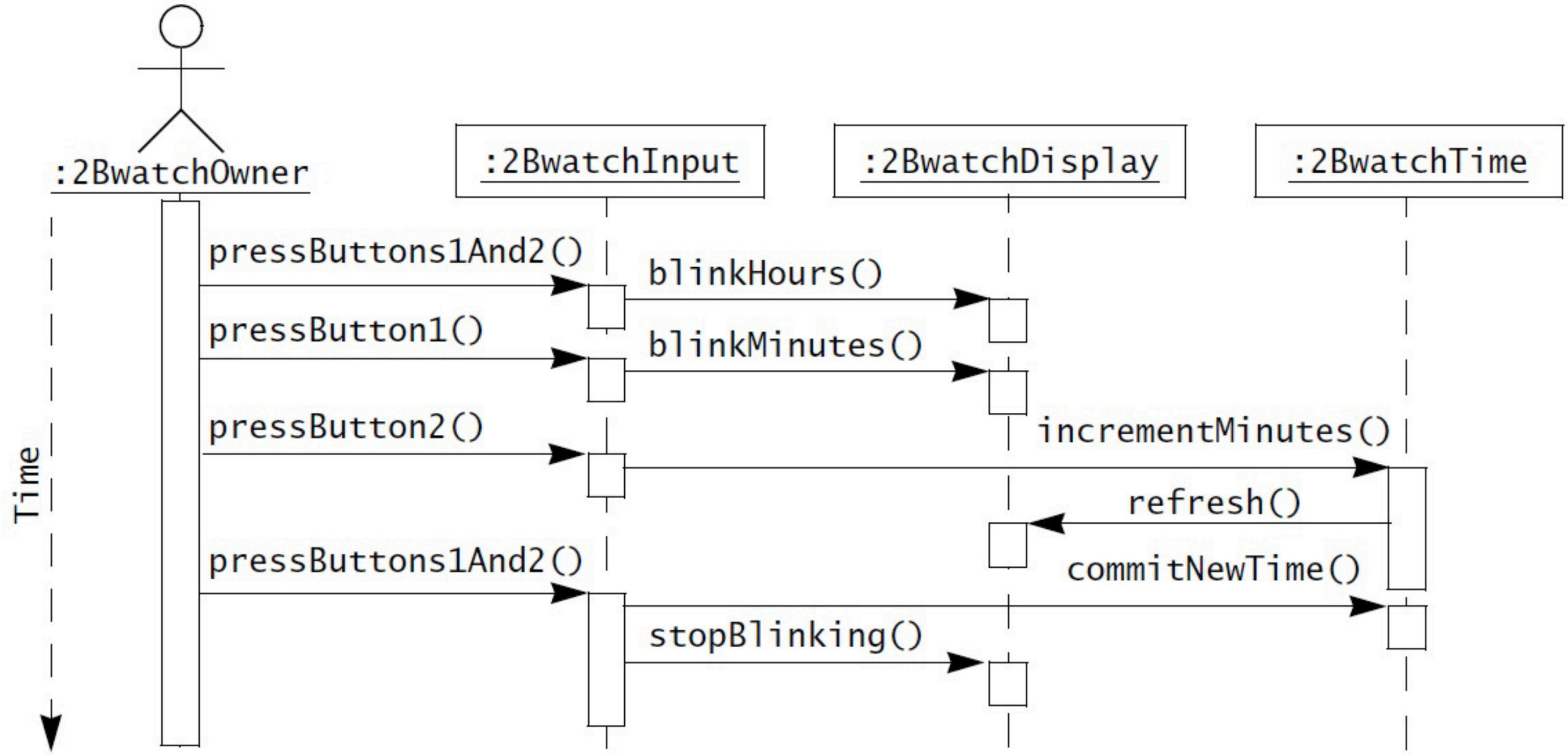
- Use case diagrams are dangerous, they are a double-edged sword as the functional decomposition they facilitate can be endless.

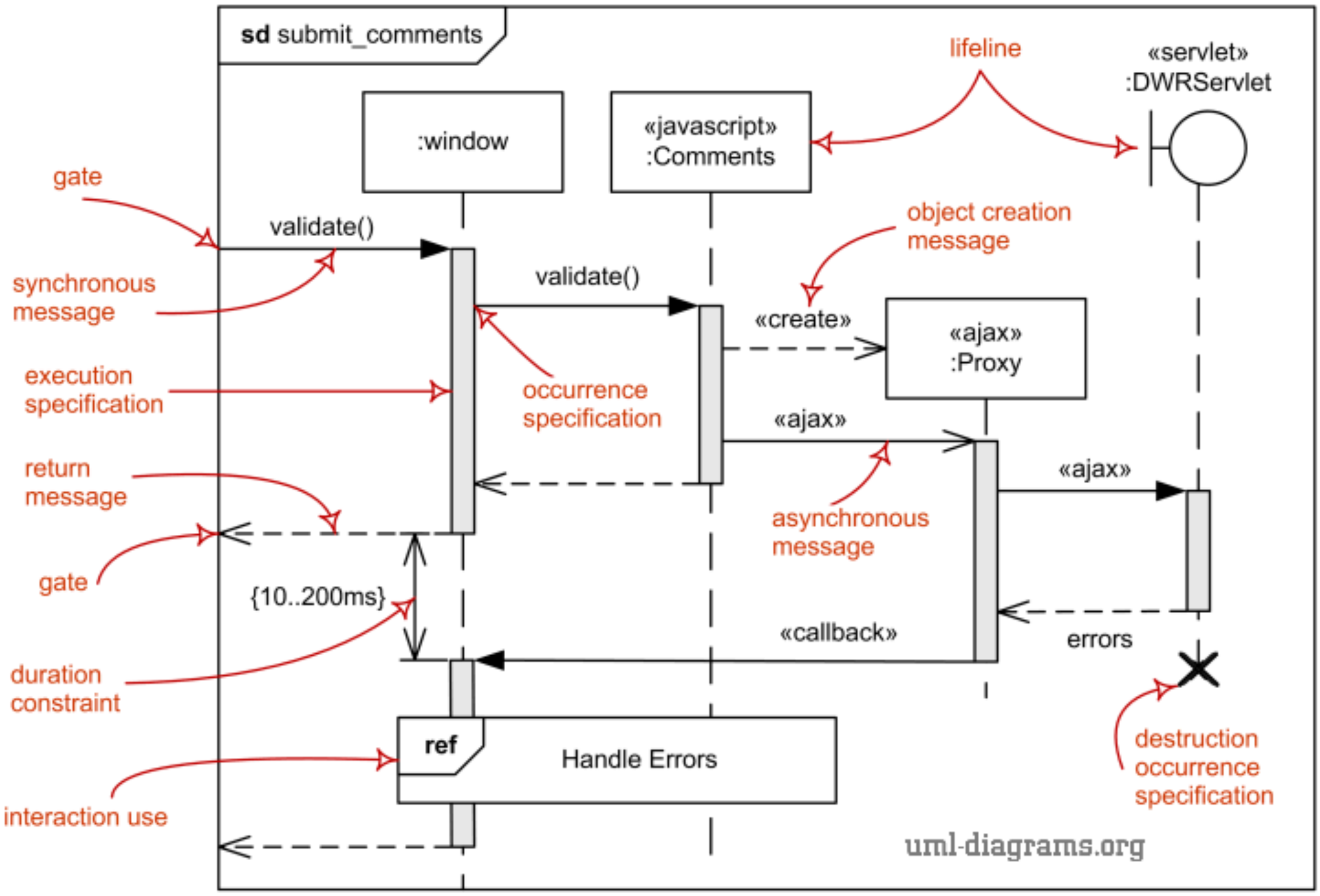
- Use cases must include also the text. The standard structure includes:
  - Name of Use Case
  - Actors: description of Actors involved in use case.
  - Entry condition: "This use case starts when…".
  - Flow of Events: free form, informal natural language.
  - Exit condition: "This use cases terminates when…".
  - Exceptions: describe what happens if things go wrong.
  - Quality Requirements: nonfunctional requirements, constraints.

# Sequence diagram



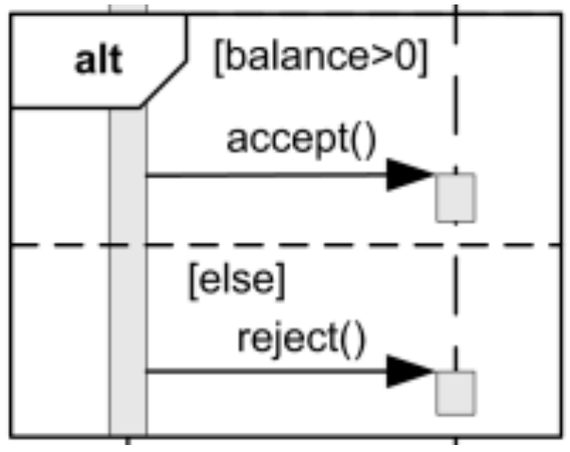

- Behaviour interaction diagrams.
- They specify the dynamic behaviour of a system as interactions between different objects.
- They capture the events that happen at runtime.
- They are "formal" and are used for specification and/or documentation.
- They focus on time and flow (e.g., concurrency) between objects.

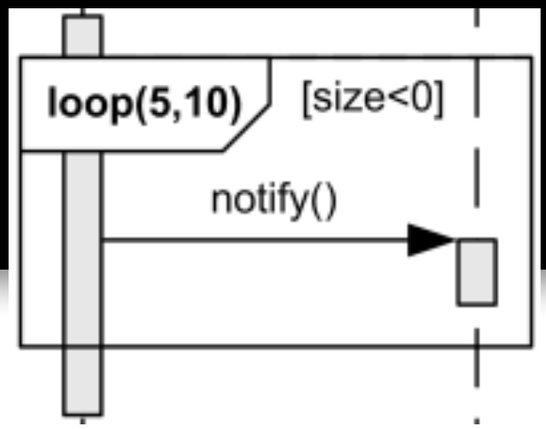

:2BwatchOwner
:2BwatchInput
:2BwatchDisplay
:2BwatchTime
pressButtons1And2()
blinkHours()
pressButton1()
blinkMinutes()
pressButton2()
incrementMinutes()
refresh()
pressButtons1And2()
commitNewTime()
stopBlinking()
Time

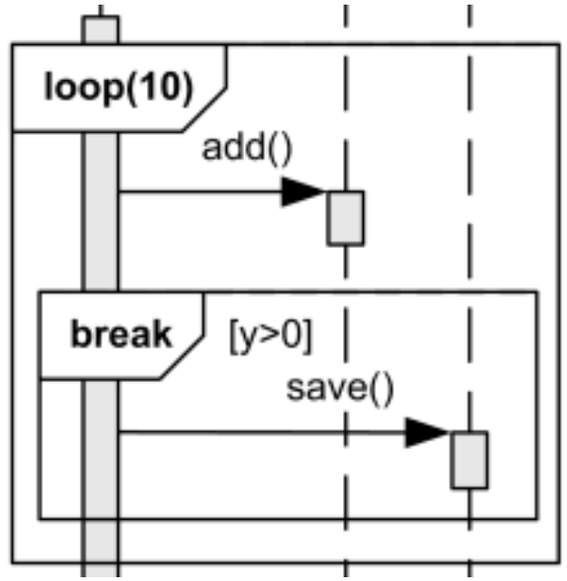

sd submit_comments
lifeline
«servlet»
:DWRServlet
:window
«javascript»
:Comments
gate
validate()
object creation
message
synchronous
message
validate()
«create»
«ajax»
:Proxy
execution
specification
occurrence
specification
«ajax»
return
message
«ajax»
asynchronous
message
gate
{10..200ms}
«callback»
errors
duration
constraint
ref
Handle Errors
destruction
occurrence
specification
interaction use
uml-diagrams.org

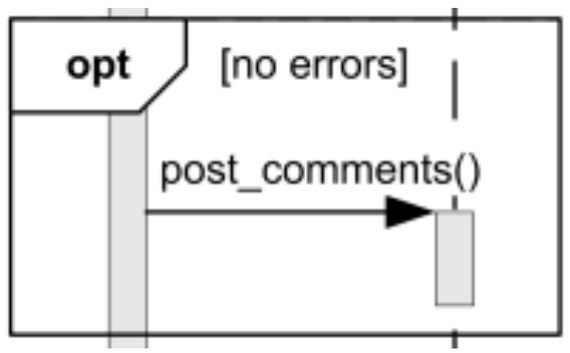
# Combined fragment

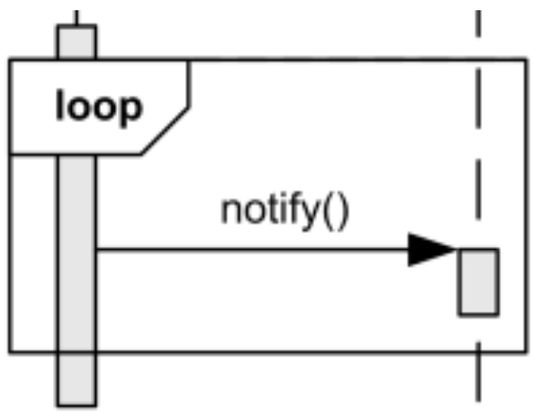
- alt - alternatives
- opt - option
- loop - iteration
- break - break
- par - parallel
- strict - strict sequencing
- seq - weak sequencing
- critical - critical region
- ignore - ignore
- consider - consider
- assert - assertion
- neg - negative

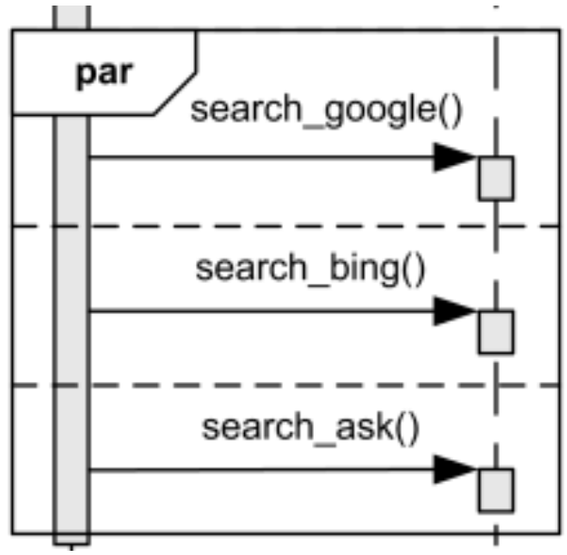


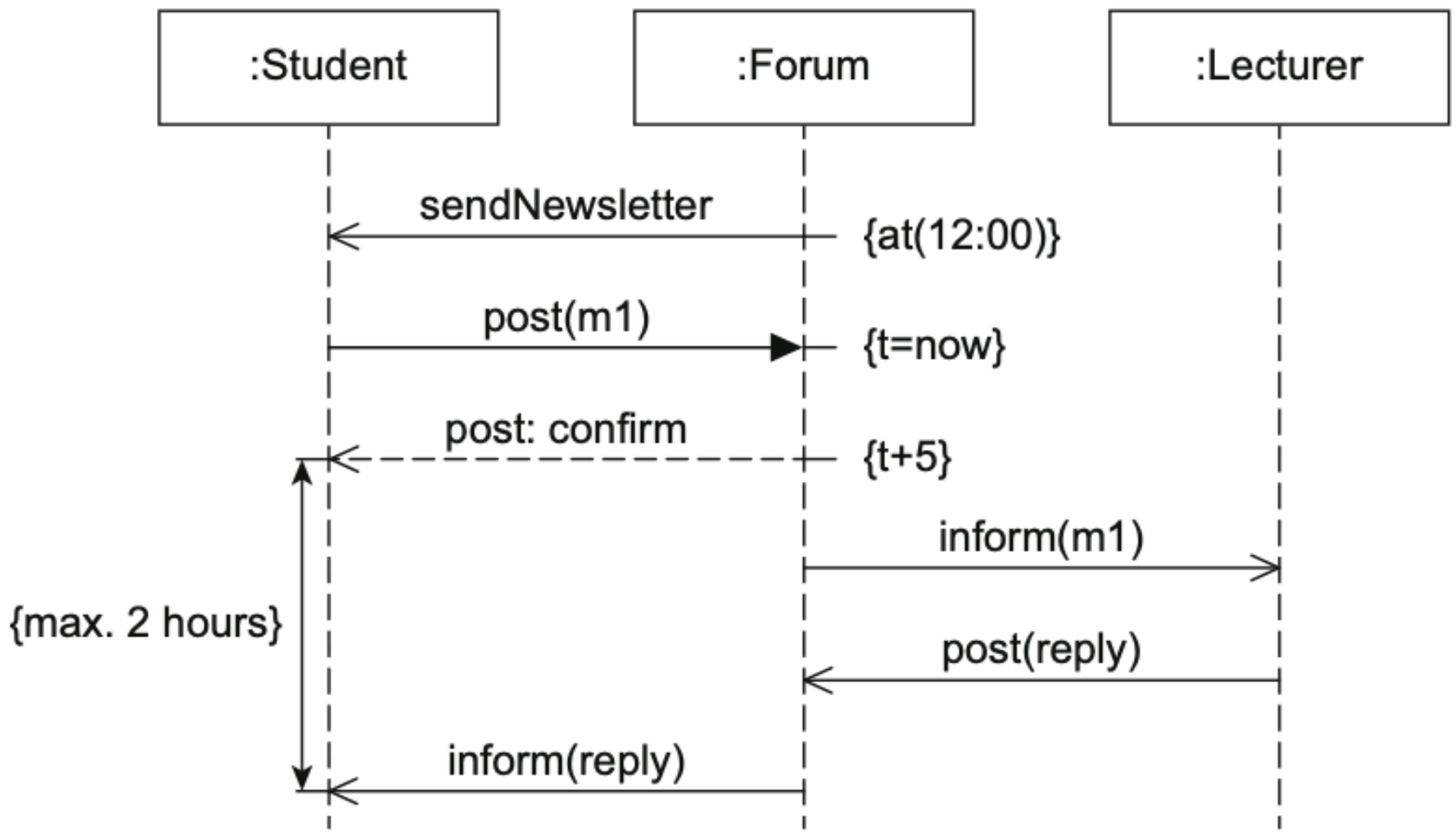



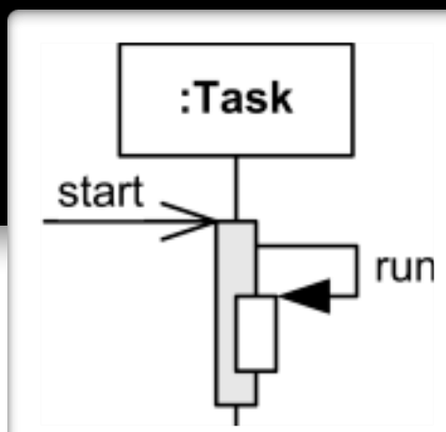
# Describing time



# Other details

- Life line = the object has been created (new).
- Execution specification or activation box = a method of the object has its activation record on the stack.
- The “return” is not always necessary. If it is obvious, it can be omitted.
- The sequence diagrams provide an immediate overview of where the methods are distributed across the object/classes.

# From past years: “What are overlapping execution specifications on the same lifeline?”




Here are some pointers to answer the question:

- Interaction partner in a sequence diagram are the rectangles and associate lifelines.
- Execution specifications are the rectangles attached to lifelines
- Execution specifications are mainly used to visualise when an interaction partner executes some behaviour.
- The interaction partner executing the behaviour can be itself. In such case the execution specification would be depicted as a rectangle contained in the previous rectangle. The meaning of the diagram below is that the interaction partner (Task), which execution was triggered by an external interaction partner by calling start, triggers a behaviour of itself by calling run.

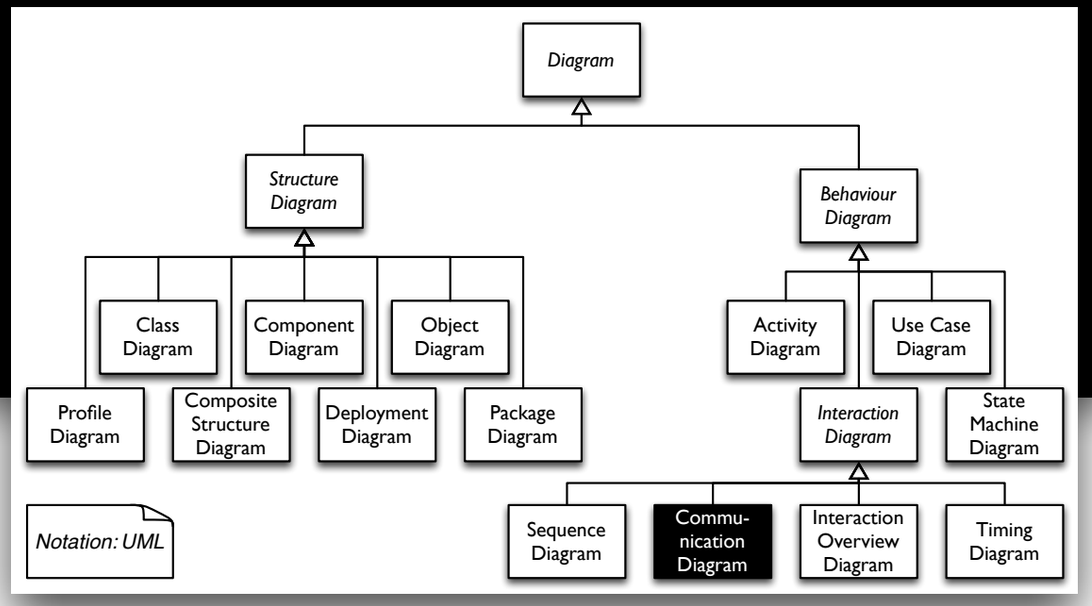
Now:

1. Execution specifications are optionals. The bit I was not sure about.
2. Overlapping execution specifications are not related to threads and such. A possible scenario might involve a modeler that wants to express that an interaction partner is executing something without being interrupted by any other messages. In the past, I have been quite verbose on sequence diagrams and expressed several important inner calls of relevant methods within the same interaction partner, which might help 'visualise' a sequence of messages. However, nowadays, I might choose a different diagram to express that goal.
3. Any call to a method from an interaction partner to itself can be shown, and if the execution specifications ought to be explicit, the overlapping execution specification is the correct syntax.
4. Another way to see this is to think about the purpose: if a modeller's intention is to make the execution specification explicit, the overlapping execution specification would be the correct syntax to use.
5. However, this bloats the diagram. Following from (1), it is allowed to avoid using the overlapping block in which case the arrow of run would hit the main execution specification.
6. Similarly and following from (1), the main execution specification could be avoided, effectively reducing the diagram to the interaction partner, the lifeline, the asynchronous start message (open arrow), and the synchronous run message (bold arrow).

As an example, in [1] the majority of sequence diagrams avoid using the execution specification in favour of clarity and use them only when necessary for improving the message conveyed to the reader. See Chapter 6 in [1] for a sample.

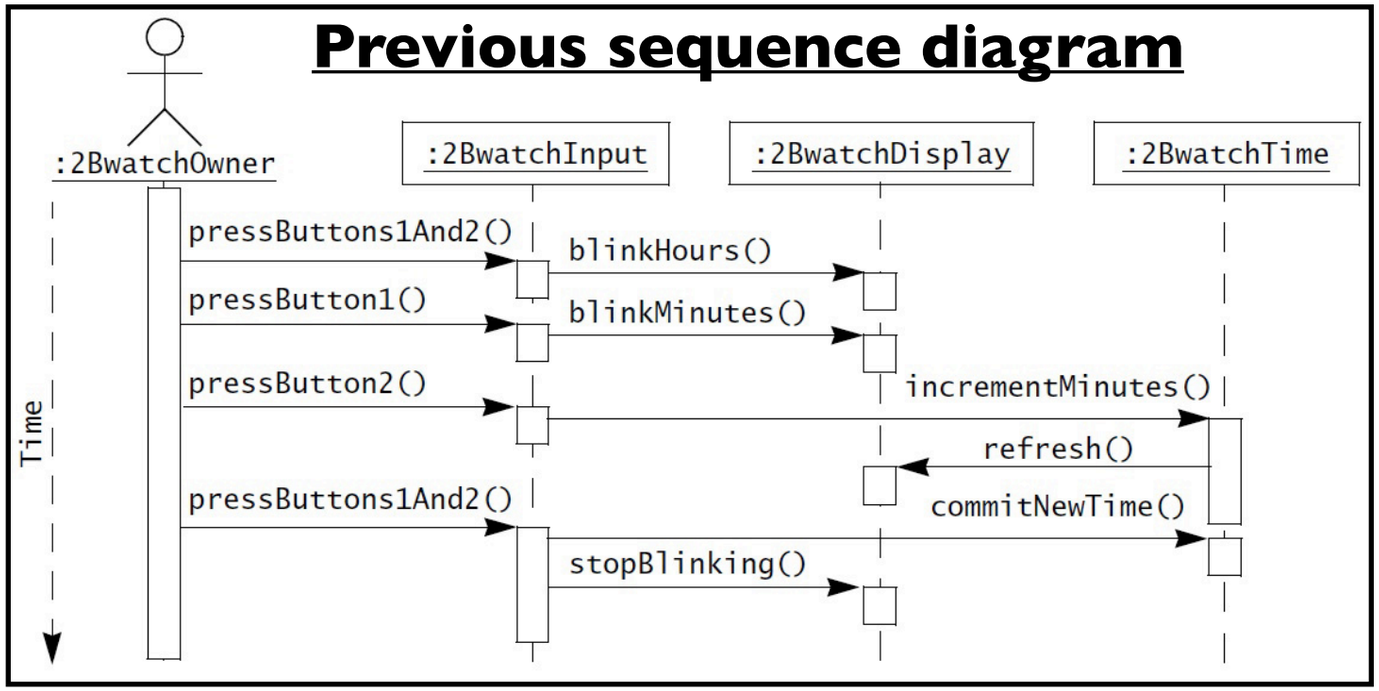
[1] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. UML@ classroom: An introduction to object-oriented modeling. Springer, 2015.

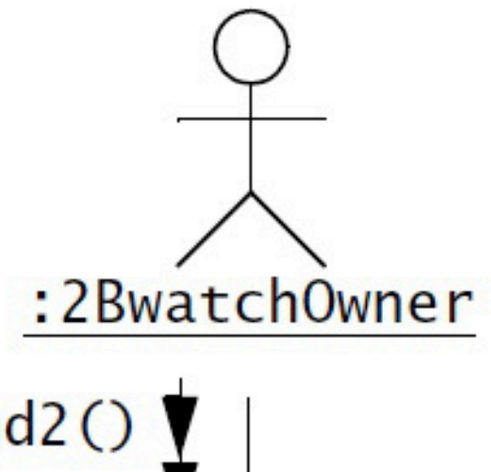
# Communication diagram

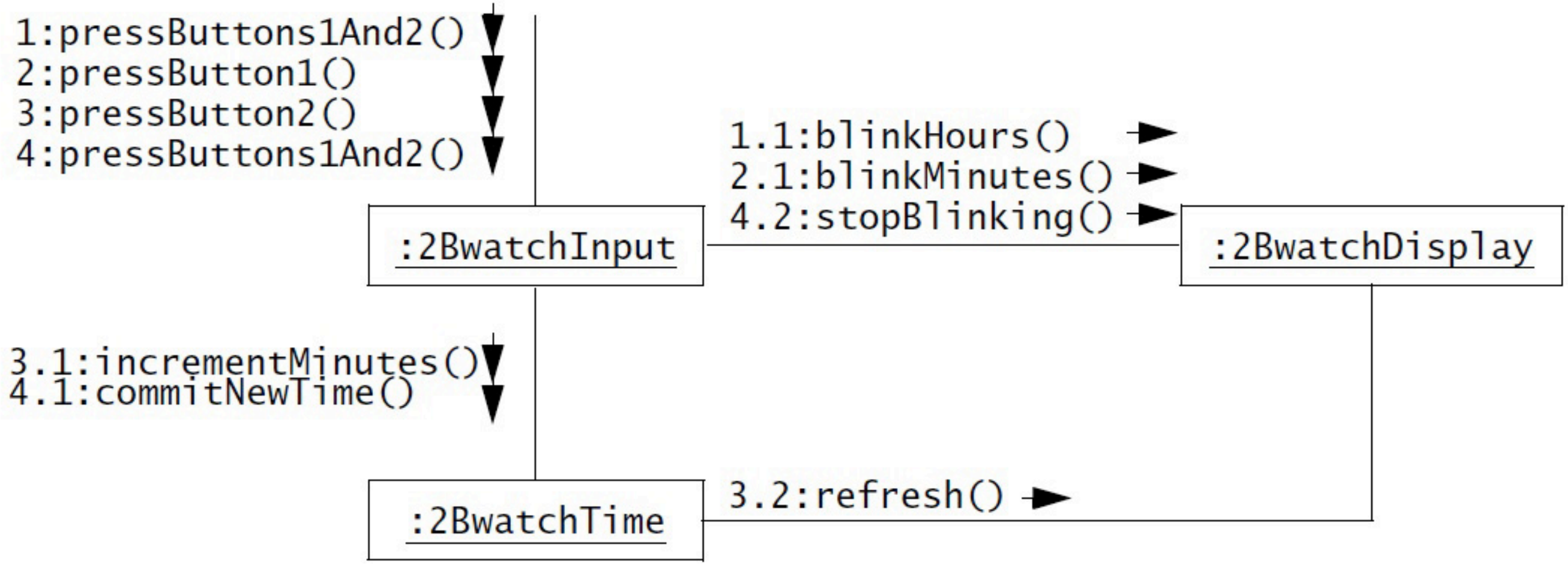


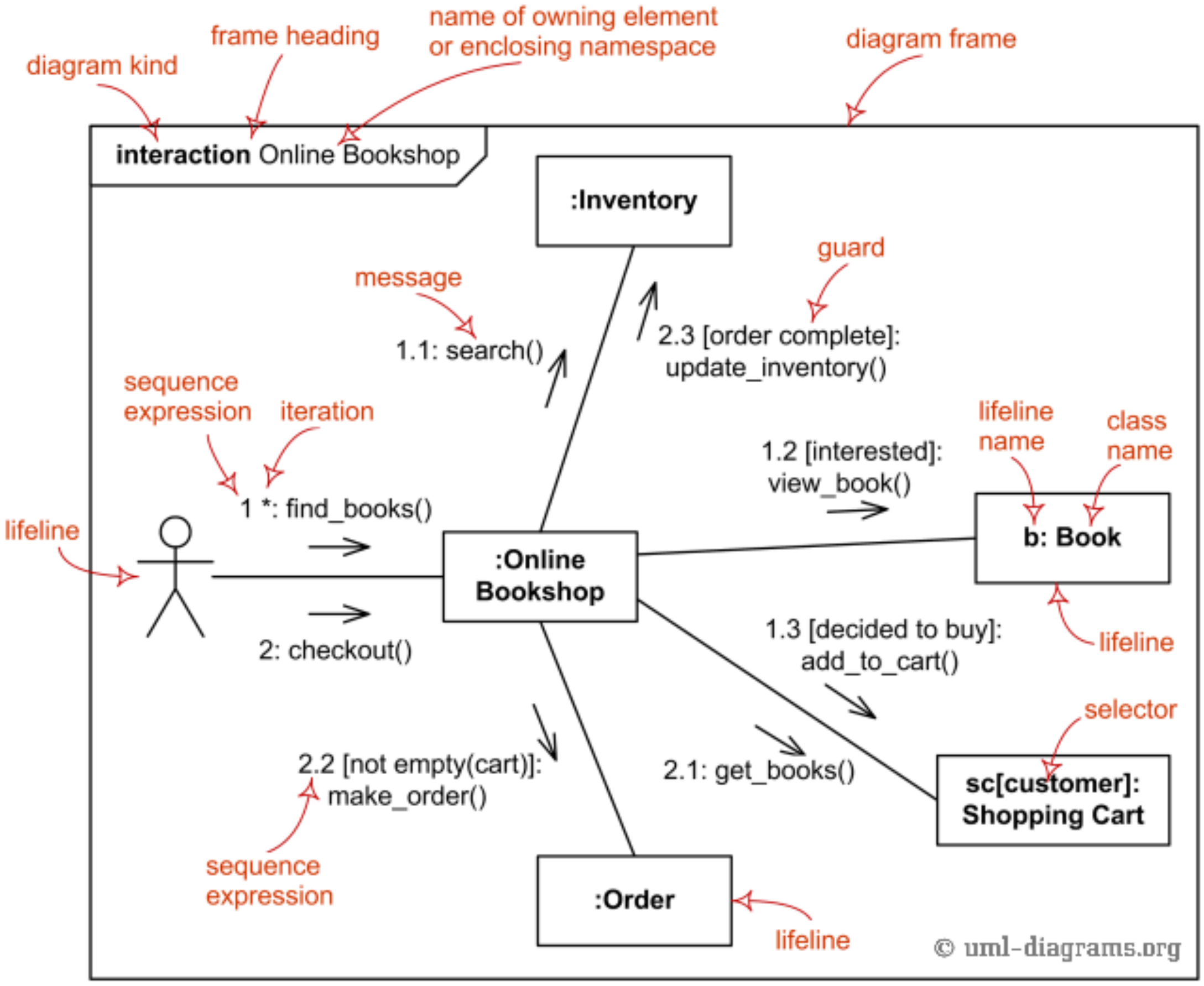

- Behaviour interaction diagrams.
- They specify the dynamic behaviour of a system as interactions between different objects.
- They contain the same information as the sequence diagrams.
- They are object diagrams + informations on the method invocations.
- They focus on the communication rather than the temporal order.
- They are more “informal” and used for sketching.

# Previous sequence diagram

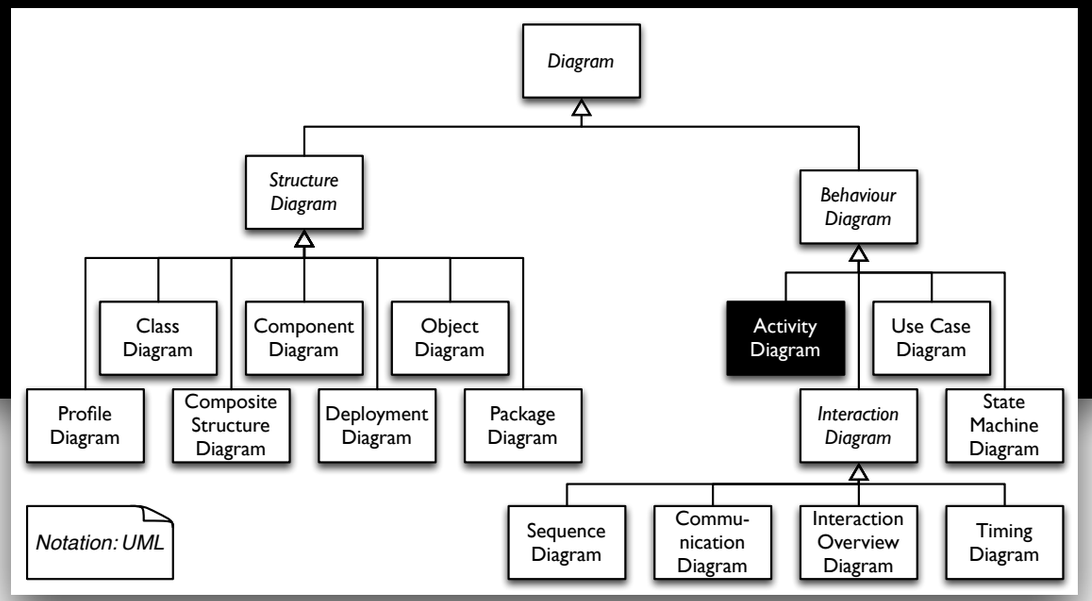


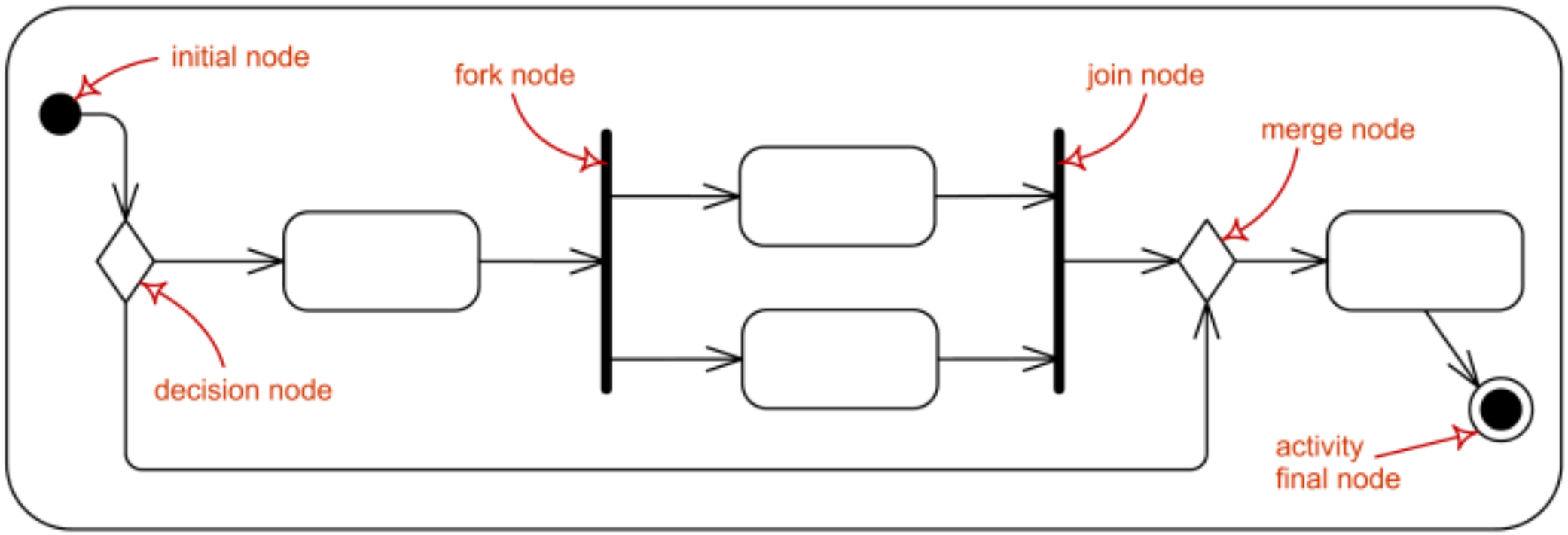


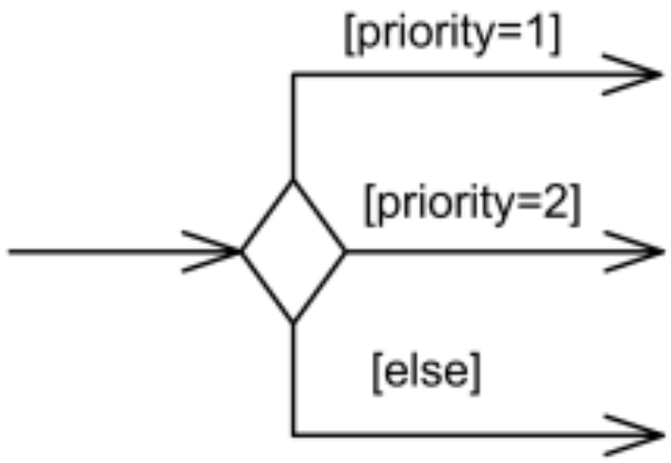
# Activity diagram

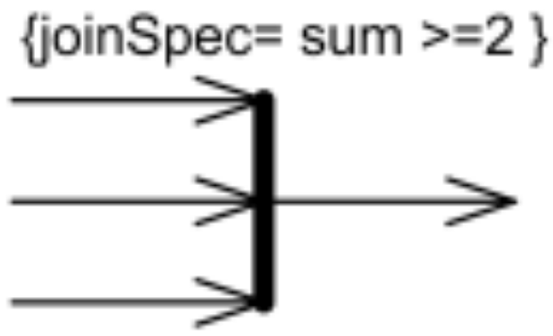



- Behaviour diagrams.
- They model the dynamic behaviour of a sub-system.
- They focus on lower level behaviour.
- They are realised on terms of one or several sequences of activities.
- Also known as flowchart.
- They include an initial and final state.
- They show decisions/merges and forks/joins.
- They can be partitioned in swim-lanes to highlight concerns.

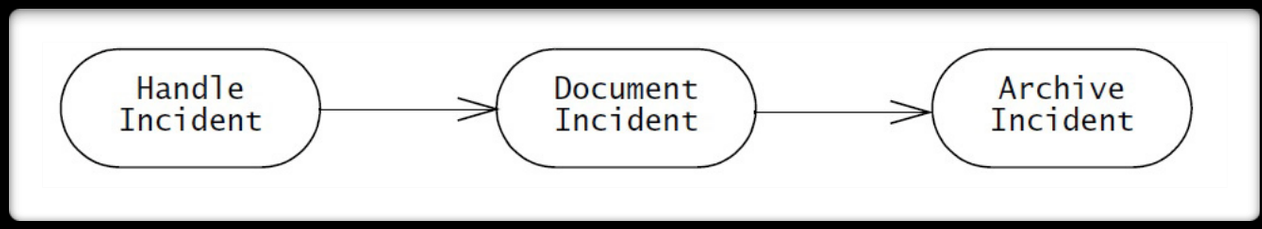
# Additional details

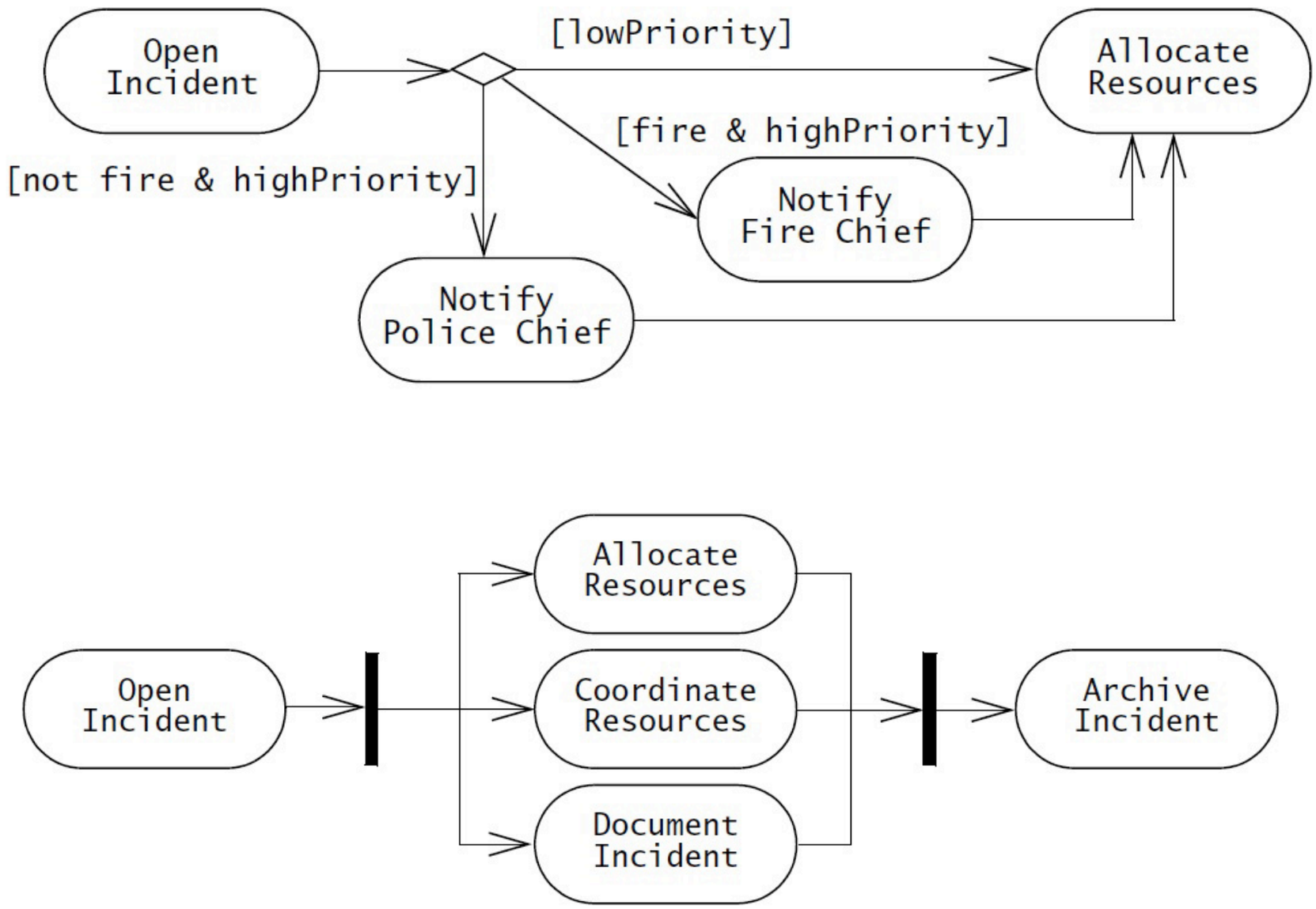


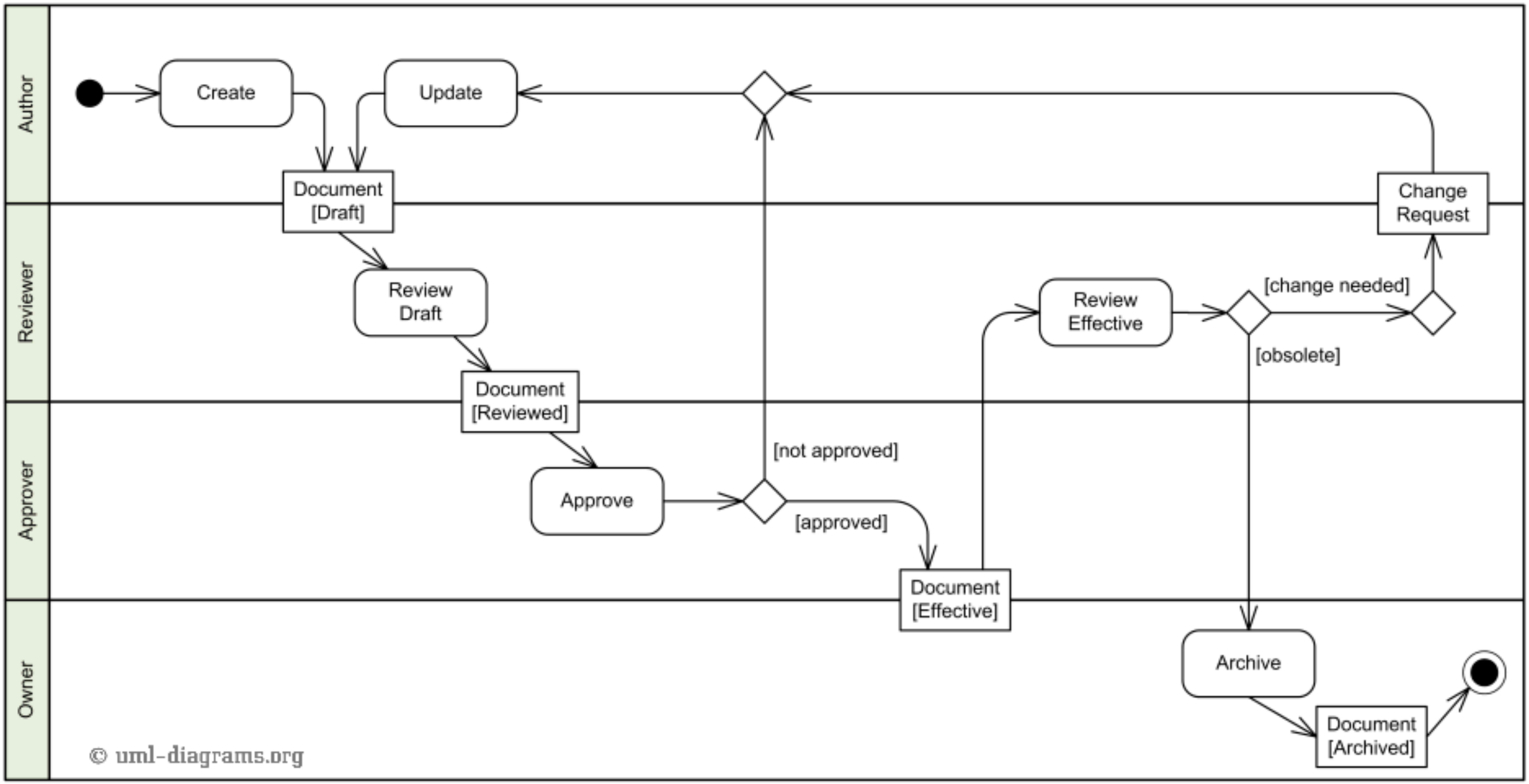



# Decisions/Merges
# Forks/Joins

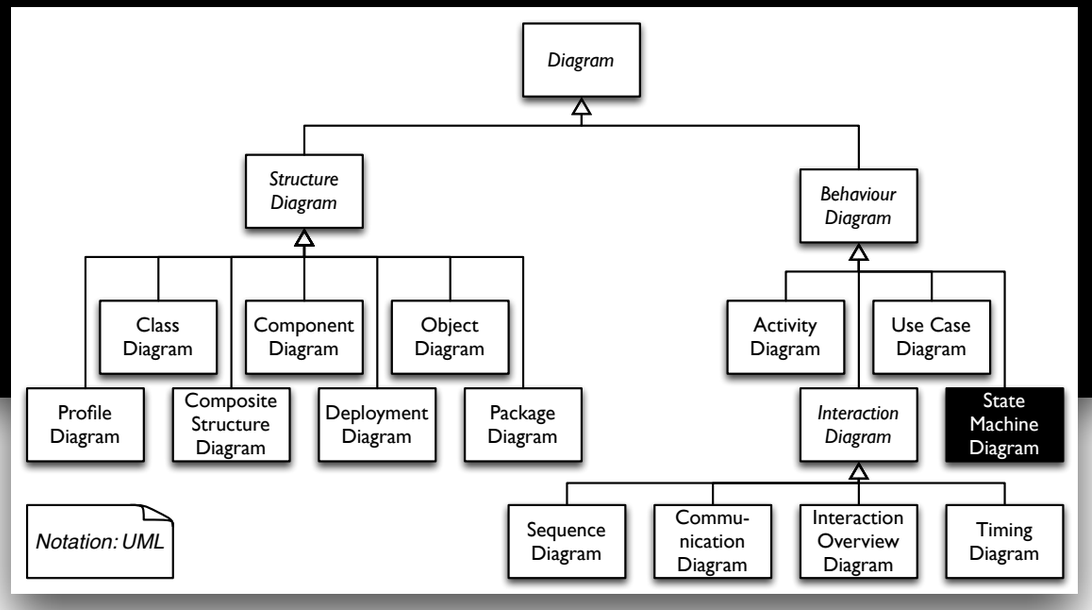


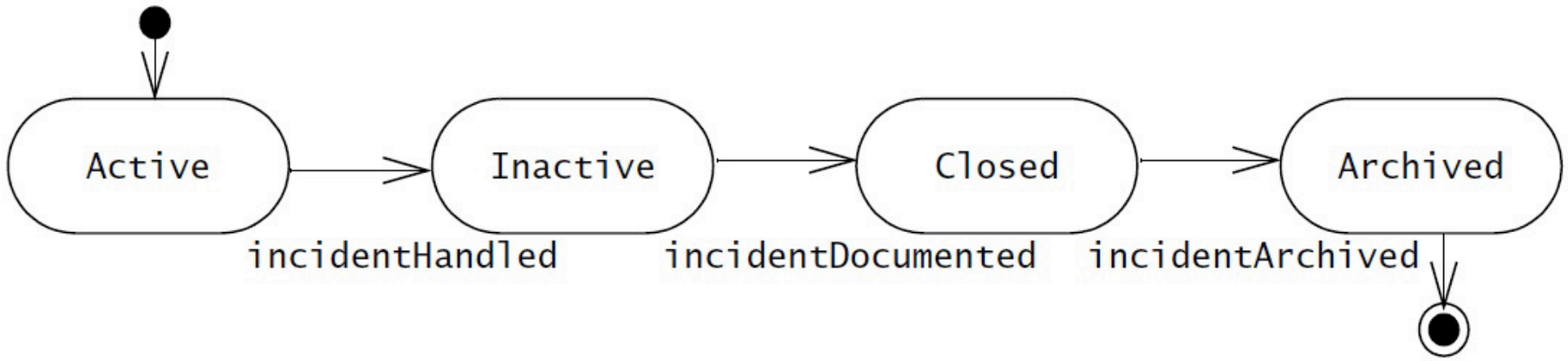
# Partitions (swimlanes)
# Object and Data tokens

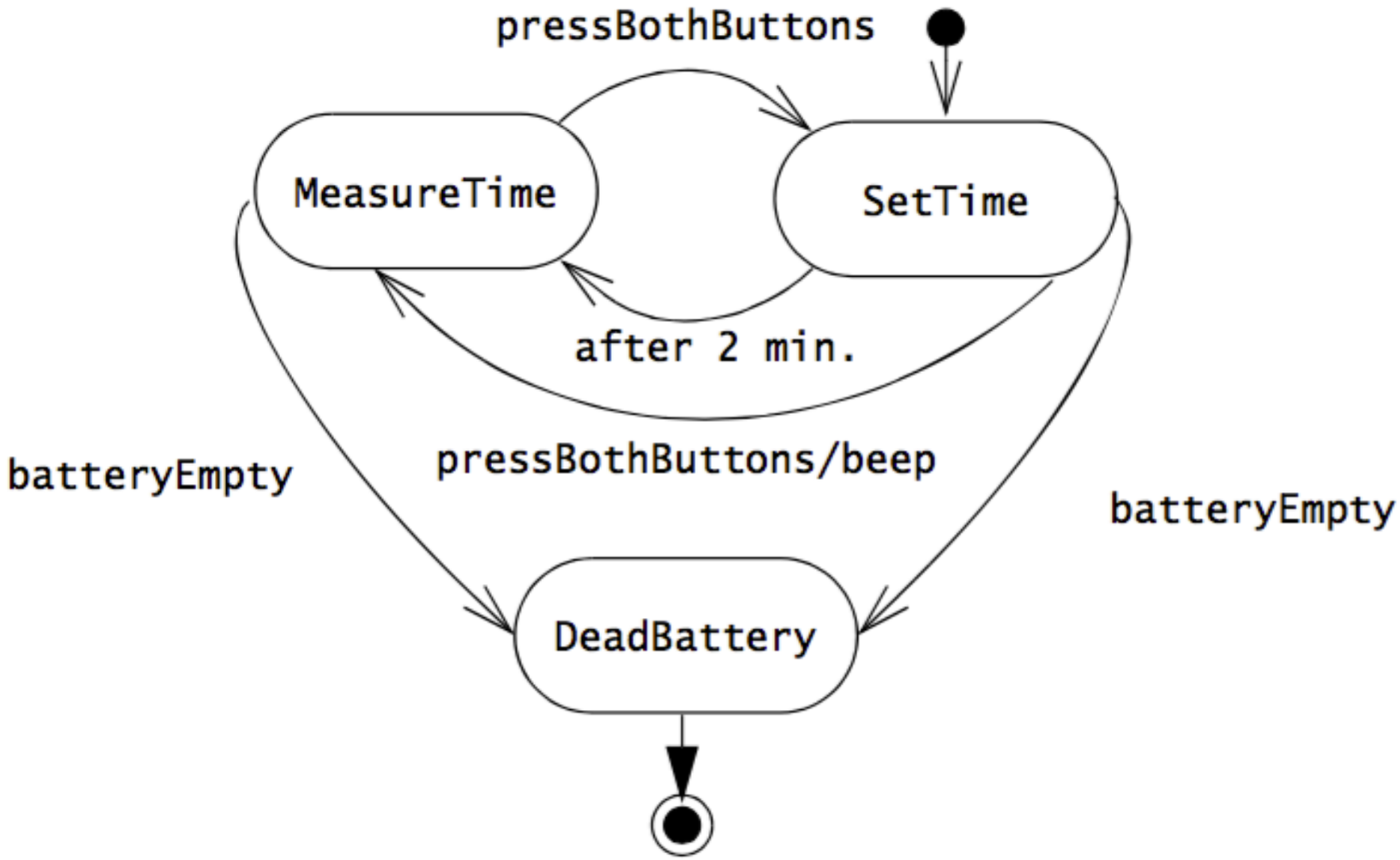


# State machine diagram

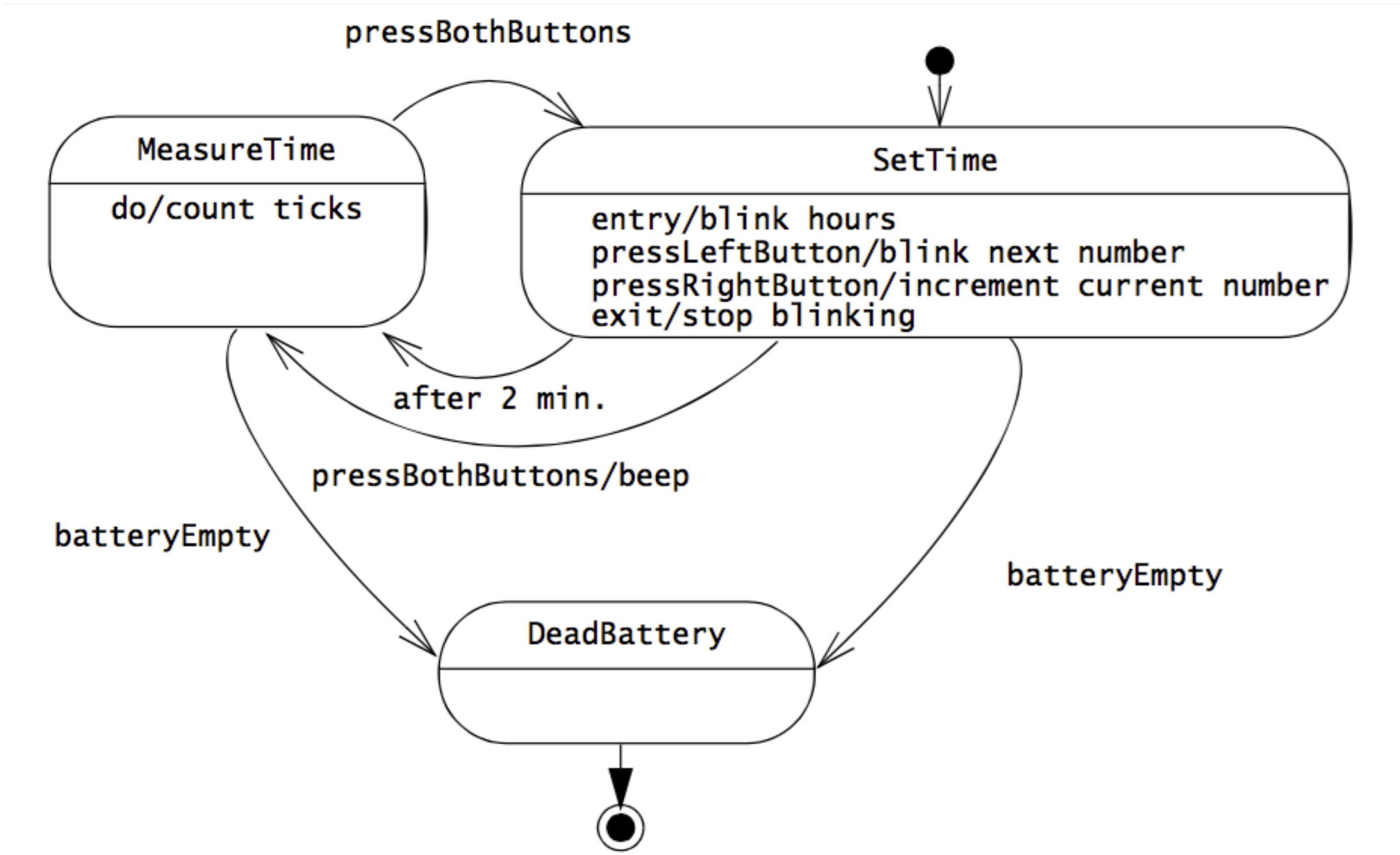



- Behaviour diagrams.
- They specify the dynamic behaviour of a single object.
- They model the sequence of states an object goes through at runtime in reaction to external events.
- They include an initial and final state.
- They show states and transactions.
- Substates are permitted.
- Transaction are labeled with events, guards, etc.

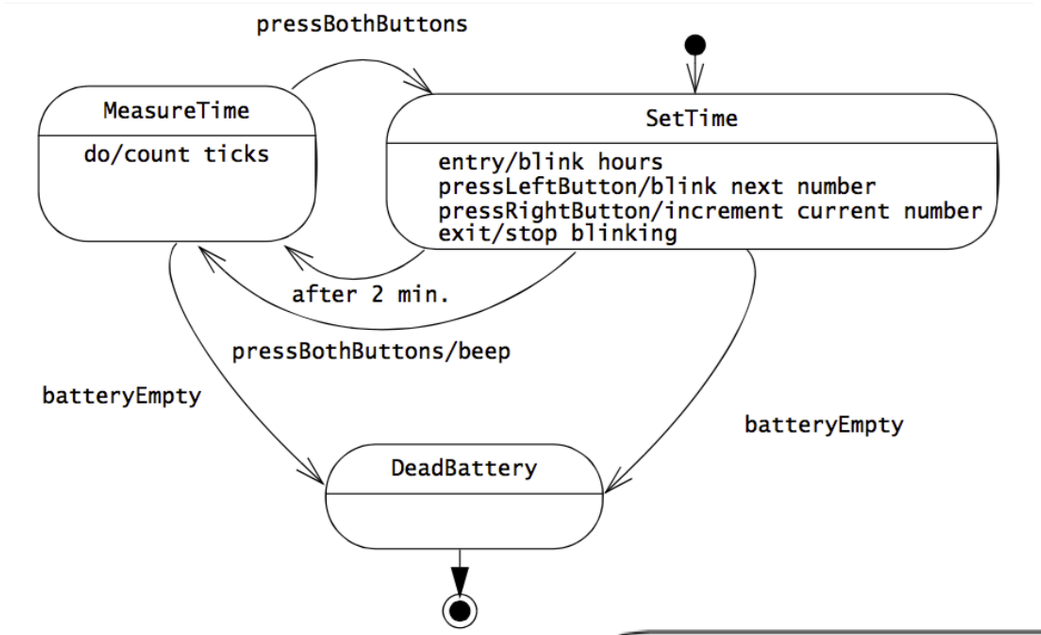


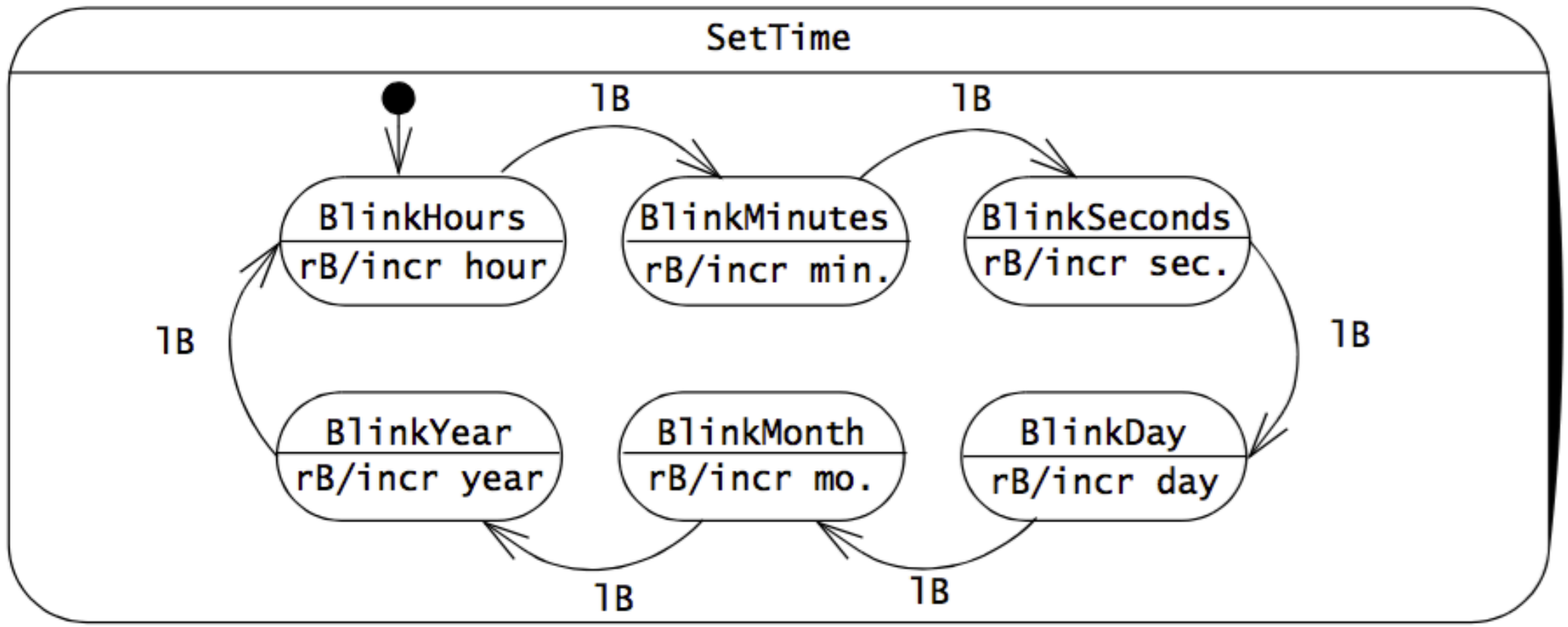



# The internal activity compartment



# Composite states (nested states)



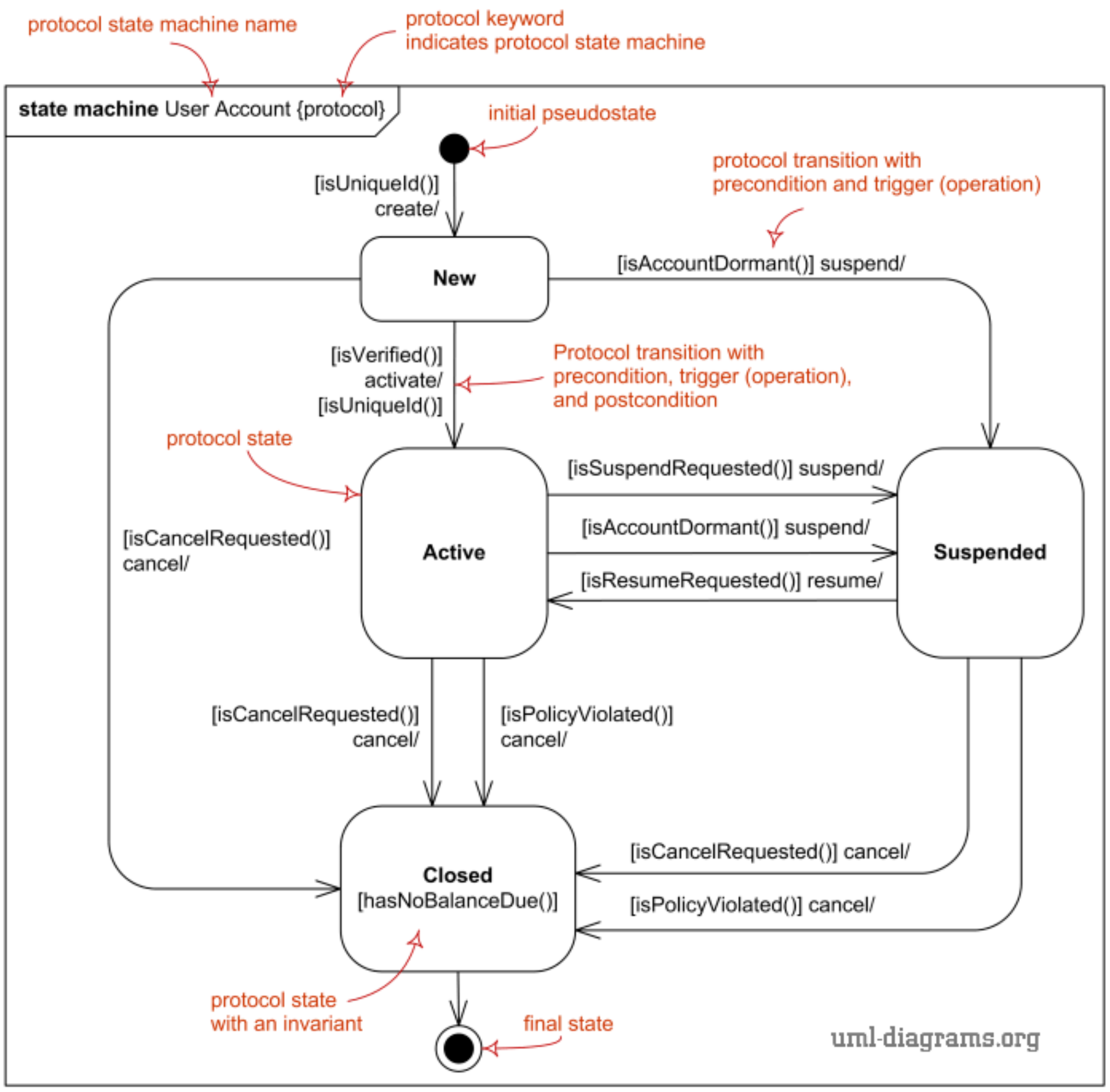



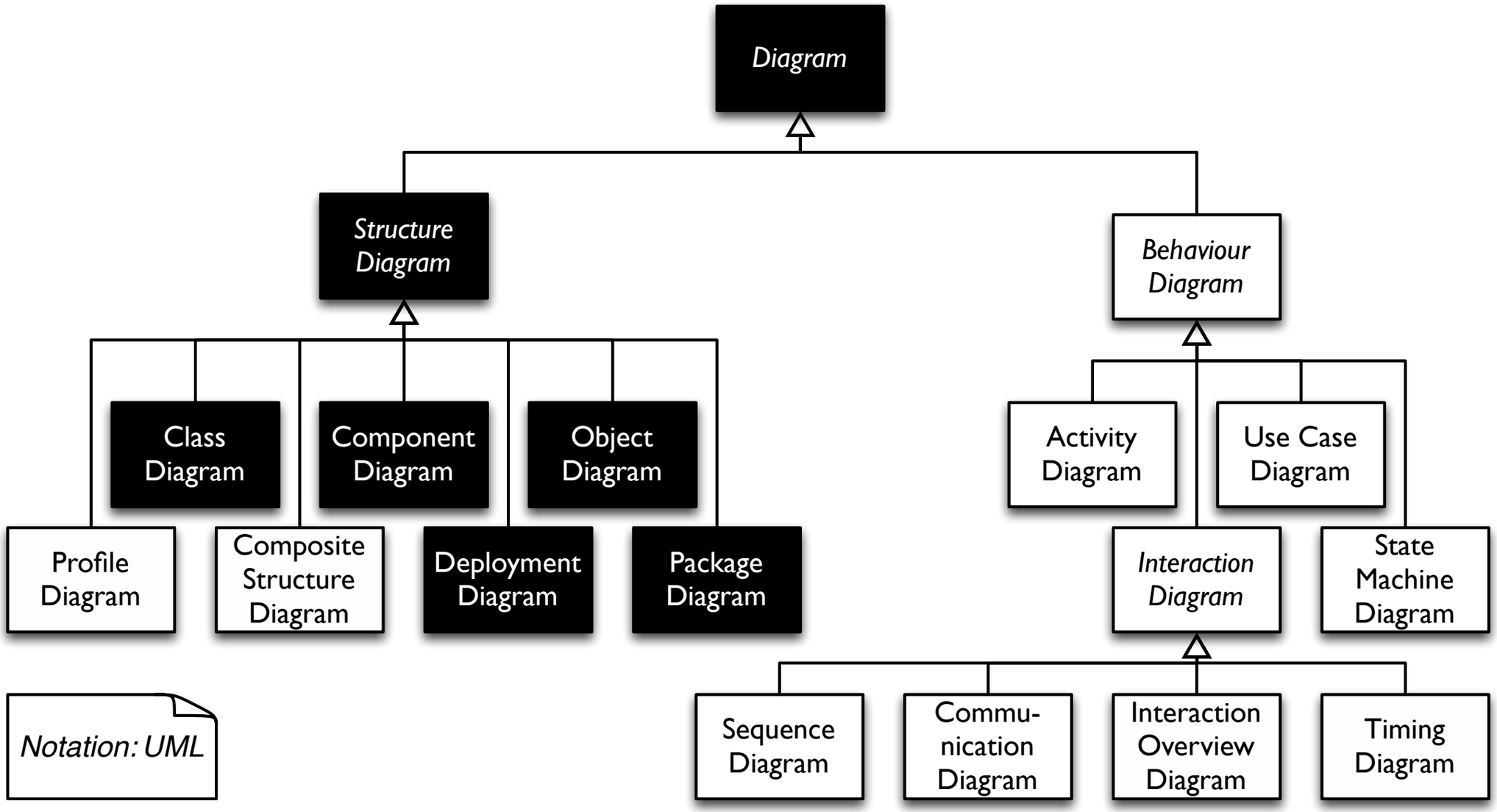
# Protocol transitions – the language of the guards

- Disclaimer: not formalised in the standard, but very useful.

protocol-transition ::= [ pre-condition ]  trigger '/' reaction  [ post-condition ]
pre-condition ::=  '[' constraint ']'
post-condition ::=  '[' constraint ']'

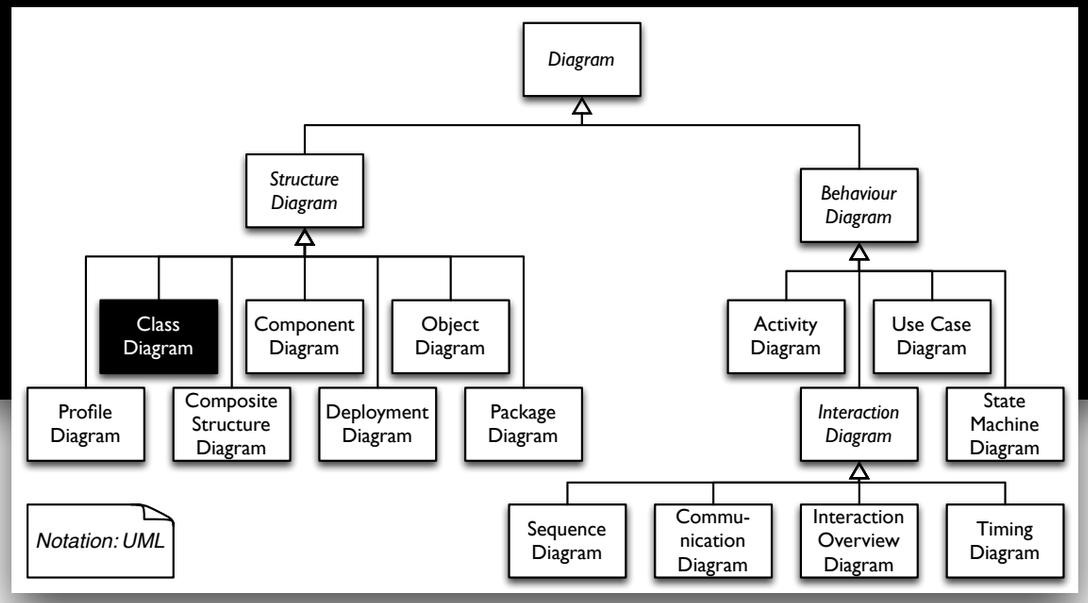
- Example: [isVerified()] activate / beep [isUniqueId()]






# Diving into UML (2nd part)

# Class diagram

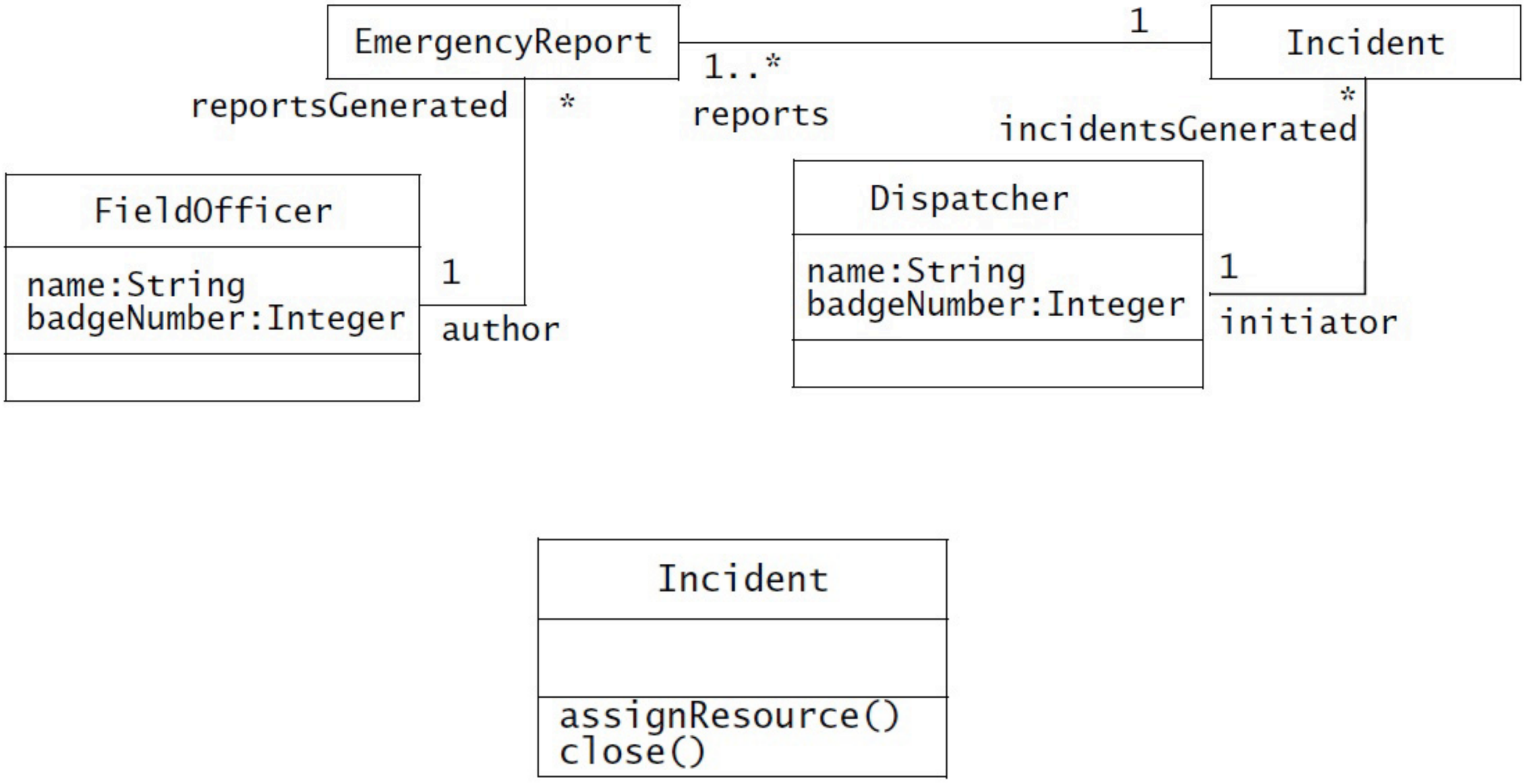


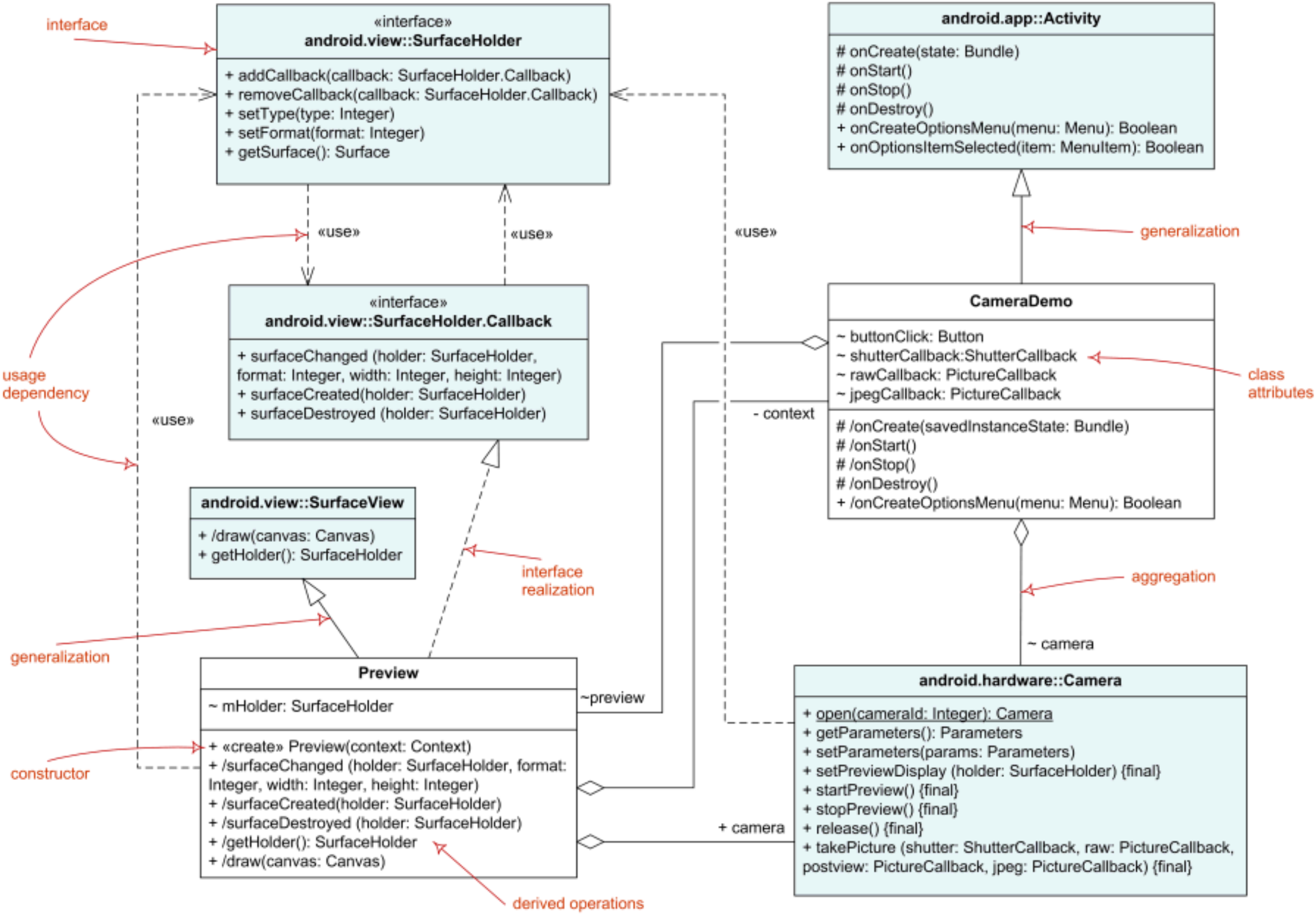

- Structure diagrams.

- Class diagrams are abstractions specifying attributes and behaviour of a set of objects.

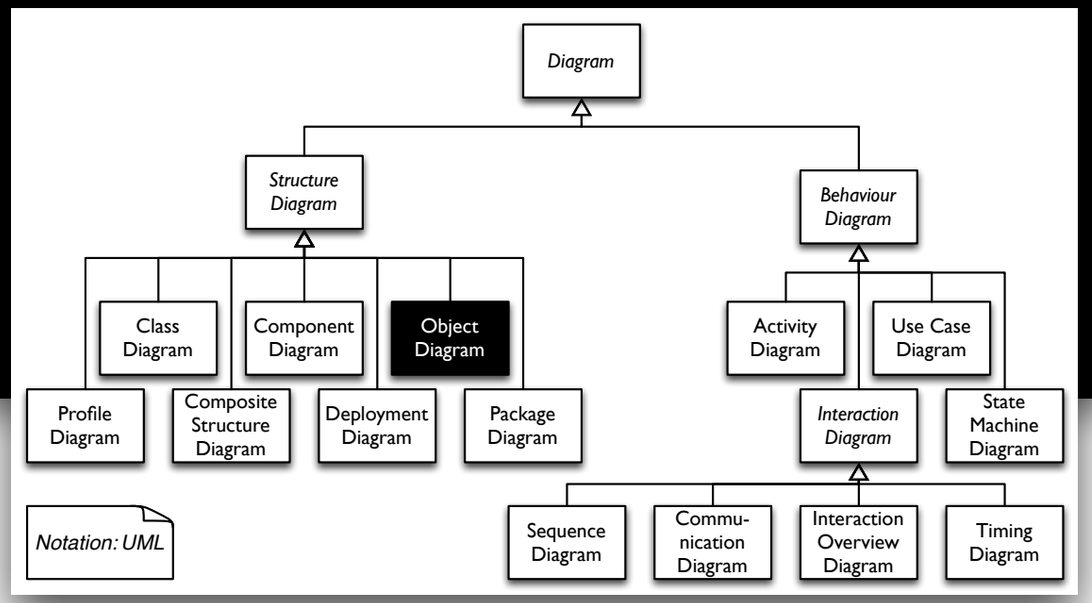
- Relationships between classes (and interfaces and packages):
  - Dependancy
  - Association
  - Aggregation/Composition
  - Inheritance

# An example


EmergencyReport
1
Incident
1..*
reportsGenerated
*
reports
*
incidentsGenerated
FieldOfficer
Dispatcher
name:String
badgeNumber:Integer
1
name:String
badgeNumber:Integer
1
author
initiator
Incident
assignResource()
close()

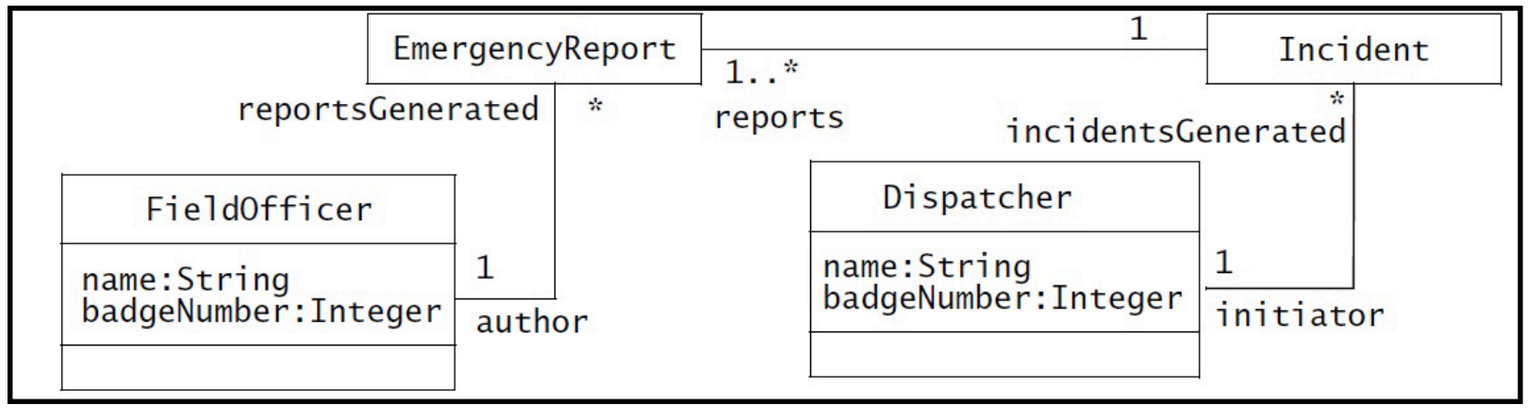


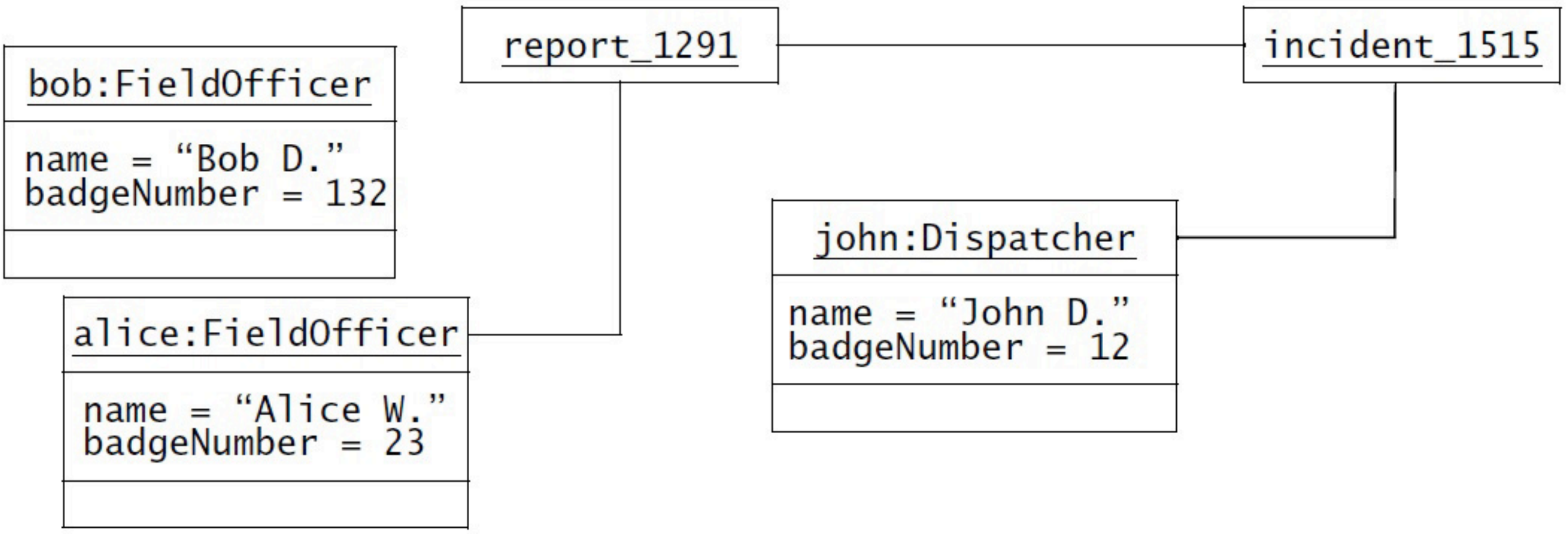
# Object diagram

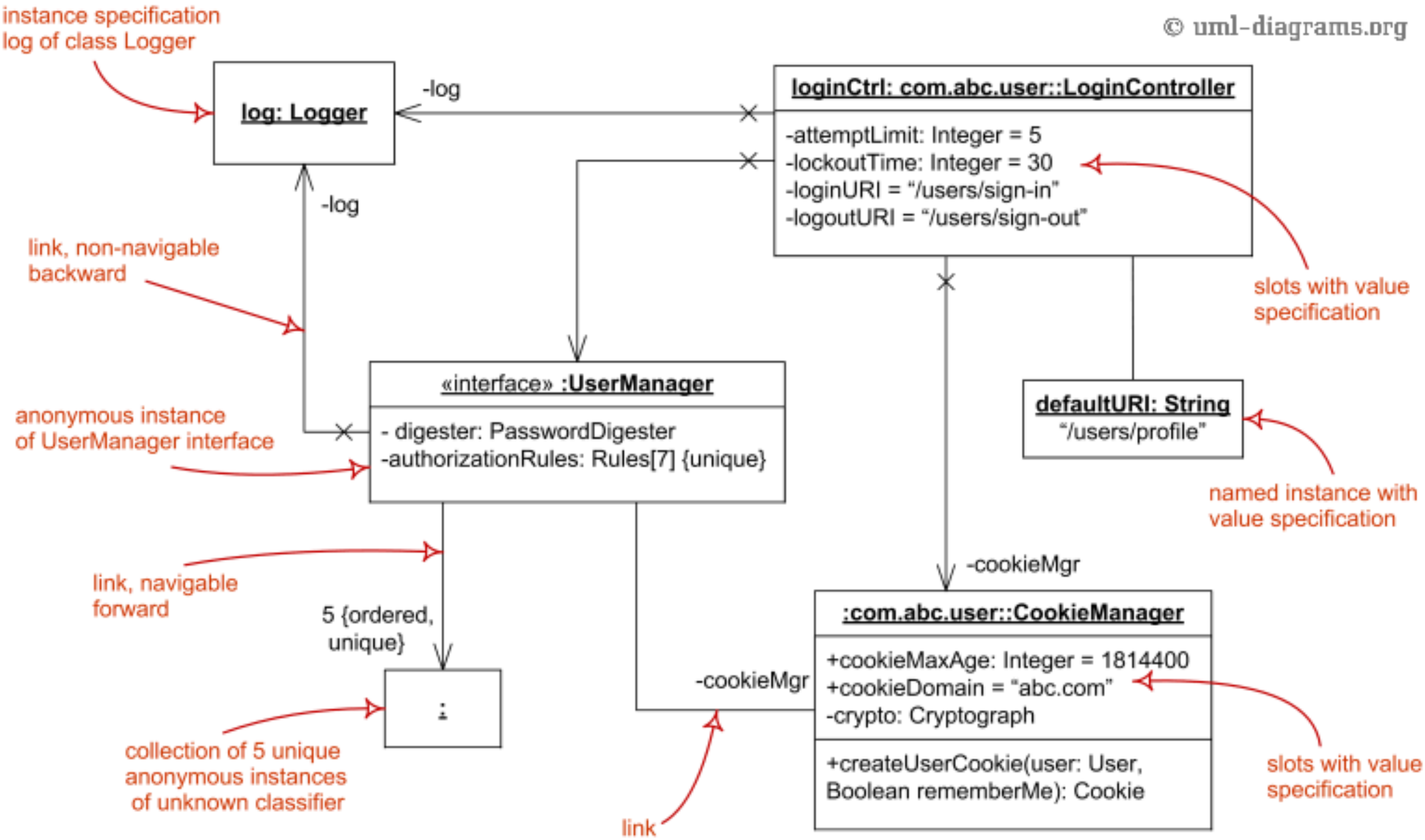



- Structure diagrams.
- Object diagrams describe the relationship between specific objects, i.e., instances of classes at a specific point in time.
- Similar to class diagrams.
  - Instances instead of classes.
  - Links instead of (instances of) associations. We will see more later.
  - Values and types of attributes describing the actual state of an instance.
  - Object names are underlined to indicate that they are instances. This rule applies in general in UML.
- They are a snapshot of a system structure at a given point in time during execution.

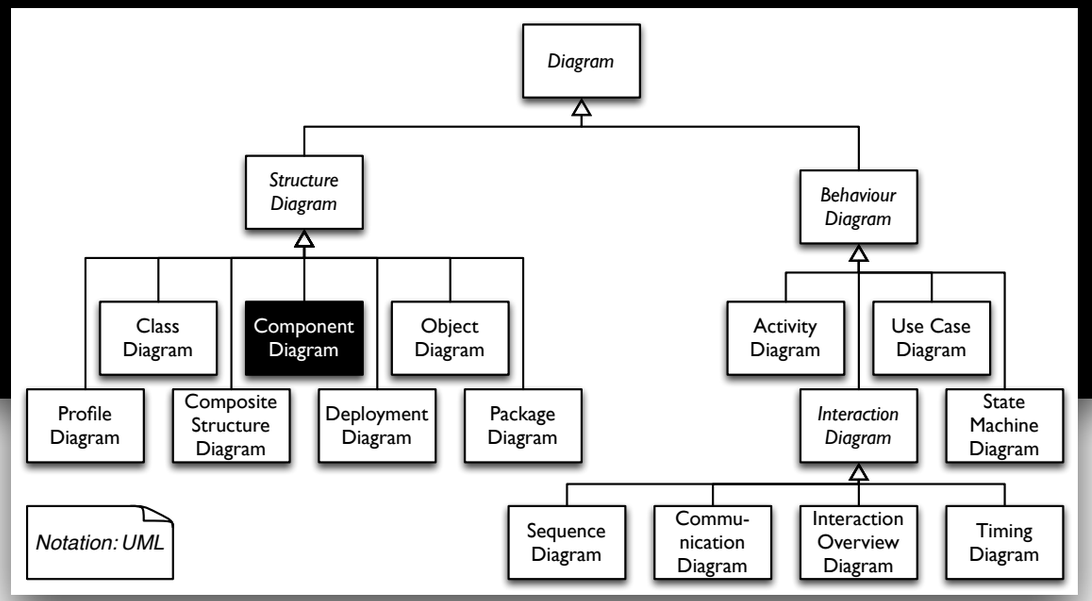
# Previous class diagram




instance specification
log of class Logger
© uml-diagrams.org
log: Logger
-log
-log
loginCtrl: com.abc.user::LoginController
-attemptLimit: Integer = 5
-lockoutTime: Integer = 30
-loginURI = "/users/sign-in"
-logoutURI = "/users/sign-out"
slots with value specification
link, non-navigable backward
«interface» :UserManager
- digester: PasswordDigester
-authorizationRules: Rules[7] {unique}
anonymous instance of UserManager interface
defaultURI: String
"/users/profile"
named instance with value specification
-cookieMgr
link, navigable forward
5 {ordered, unique}
:
collection of 5 unique anonymous instances of unknown classifier
-cookieMgr
:com.abc.user::CookieManager
+cookieMaxAge: Integer = 1814400
+cookieDomain = "abc.com"
-crypto: Cryptograph
+createUserCookie(user: User, Boolean rememberMe): Cookie
slots with value specification
link

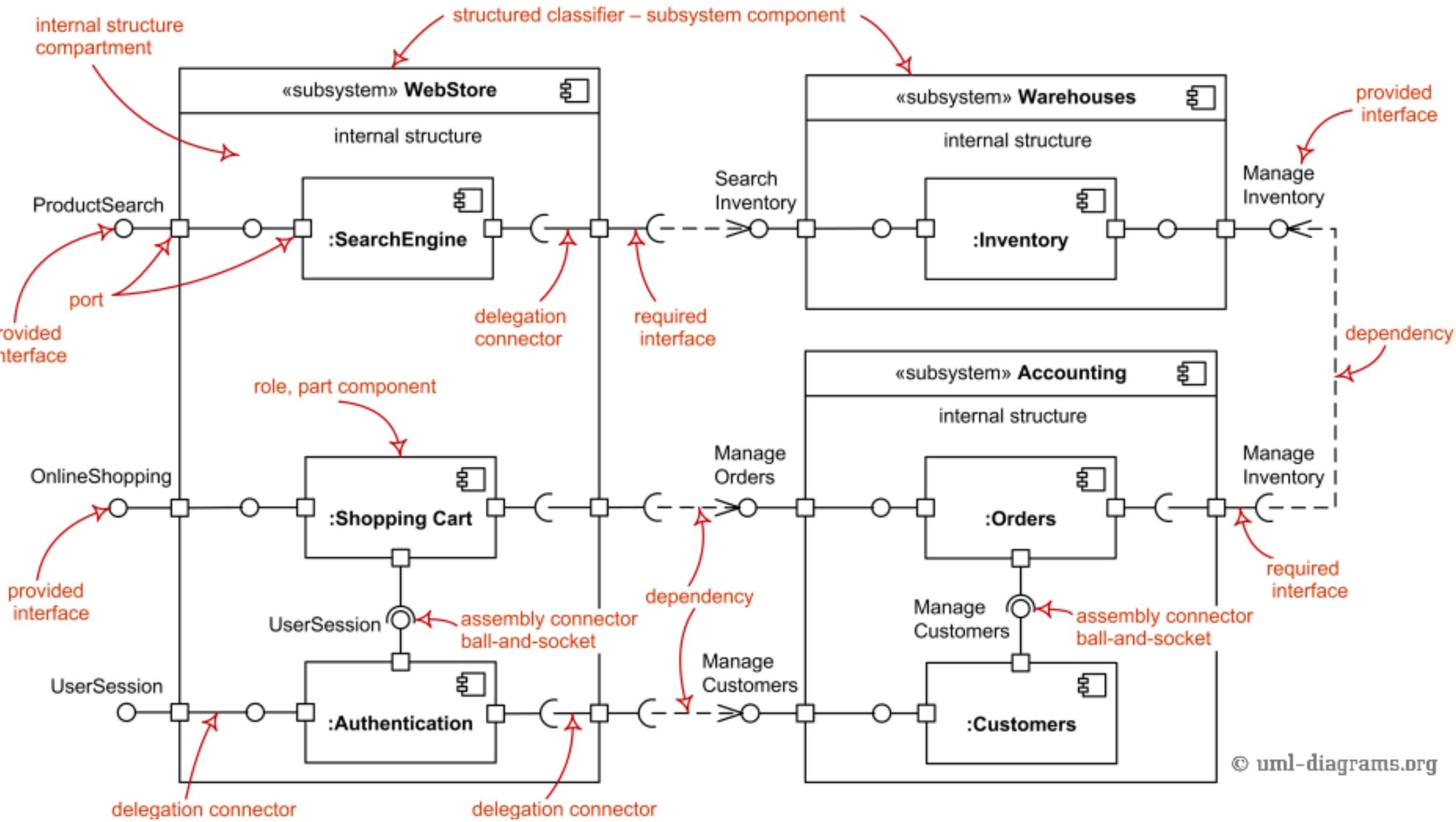
# Component diagram



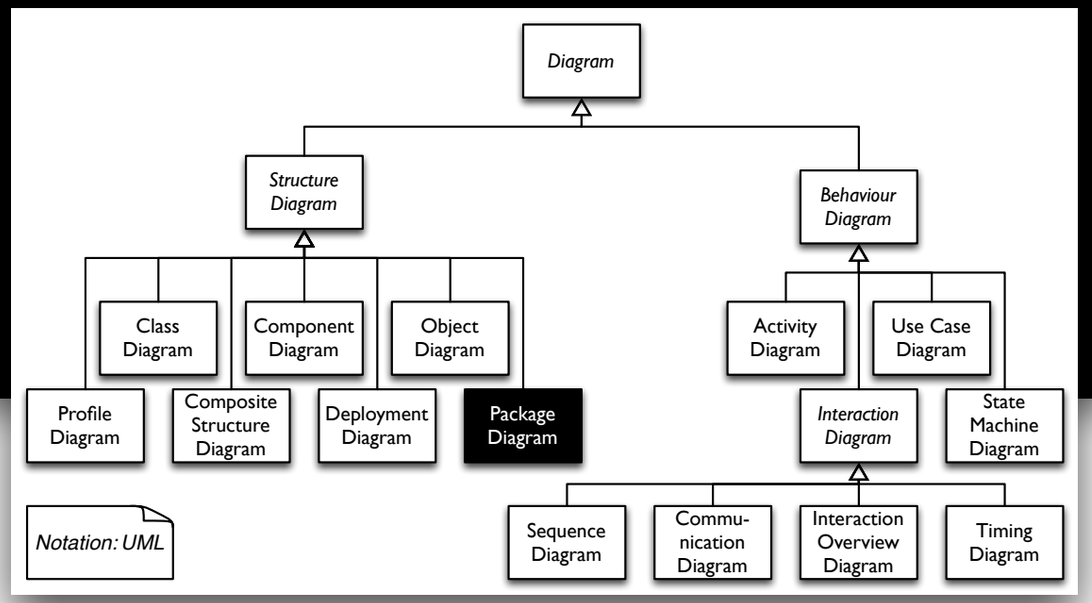

- Structure diagrams.

- Component diagram shows components, provided and required interfaces, ports, and relationships between them. This type of diagrams is heavily used in Component-Based Development (CBD) to describe systems with Service-Oriented Architecture (SOA).

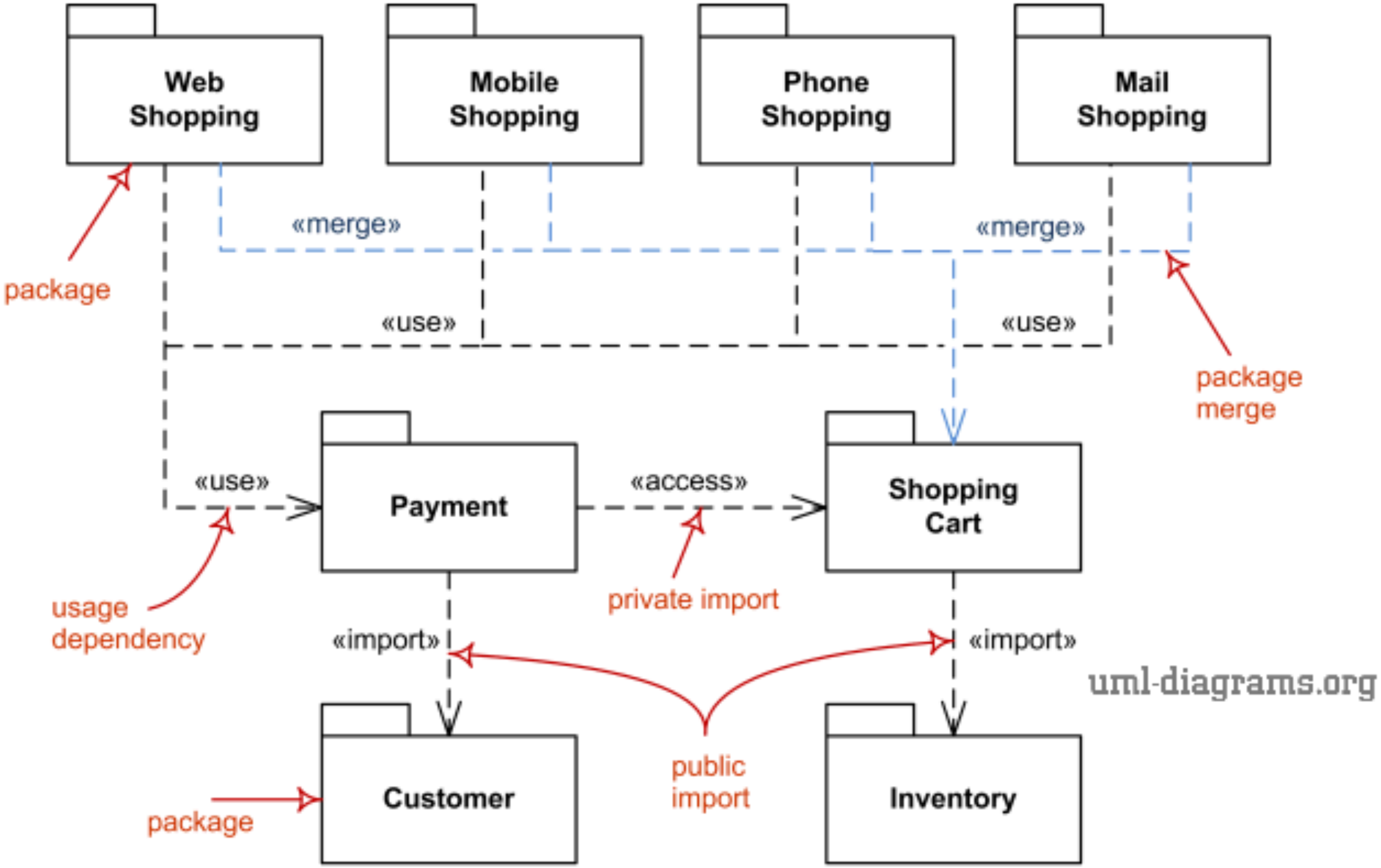
- Components in UML could represent
  - logical components (e.g., business components, process components), and
  - physical components (e.g., CORBA components, EJB components, .NET components, WSDL components, etc.),
  - along with the artifacts that implement them and the nodes on which they are deployed and executed.


internal structure compartment
structured classifier – subsystem component
«subsystem» WebStore
internal structure
ProductSearch
:SearchEngine
port
provided interface
delegation connector
required interface
Search Inventory
«subsystem» Warehouses
internal structure
:Inventory
provided interface
Manage Inventory
dependency
role, part component
«subsystem» Accounting
internal structure
OnlineShopping
:Shopping Cart
Manage Orders
:Orders
Manage Inventory
provided interface
dependency
required interface
UserSession
assembly connector ball-and-socket
Manage Customers
assembly connector ball-and-socket
UserSession
:Authentication
Manage Customers
:Customers
delegation connector
delegation connector
© uml-diagrams.org

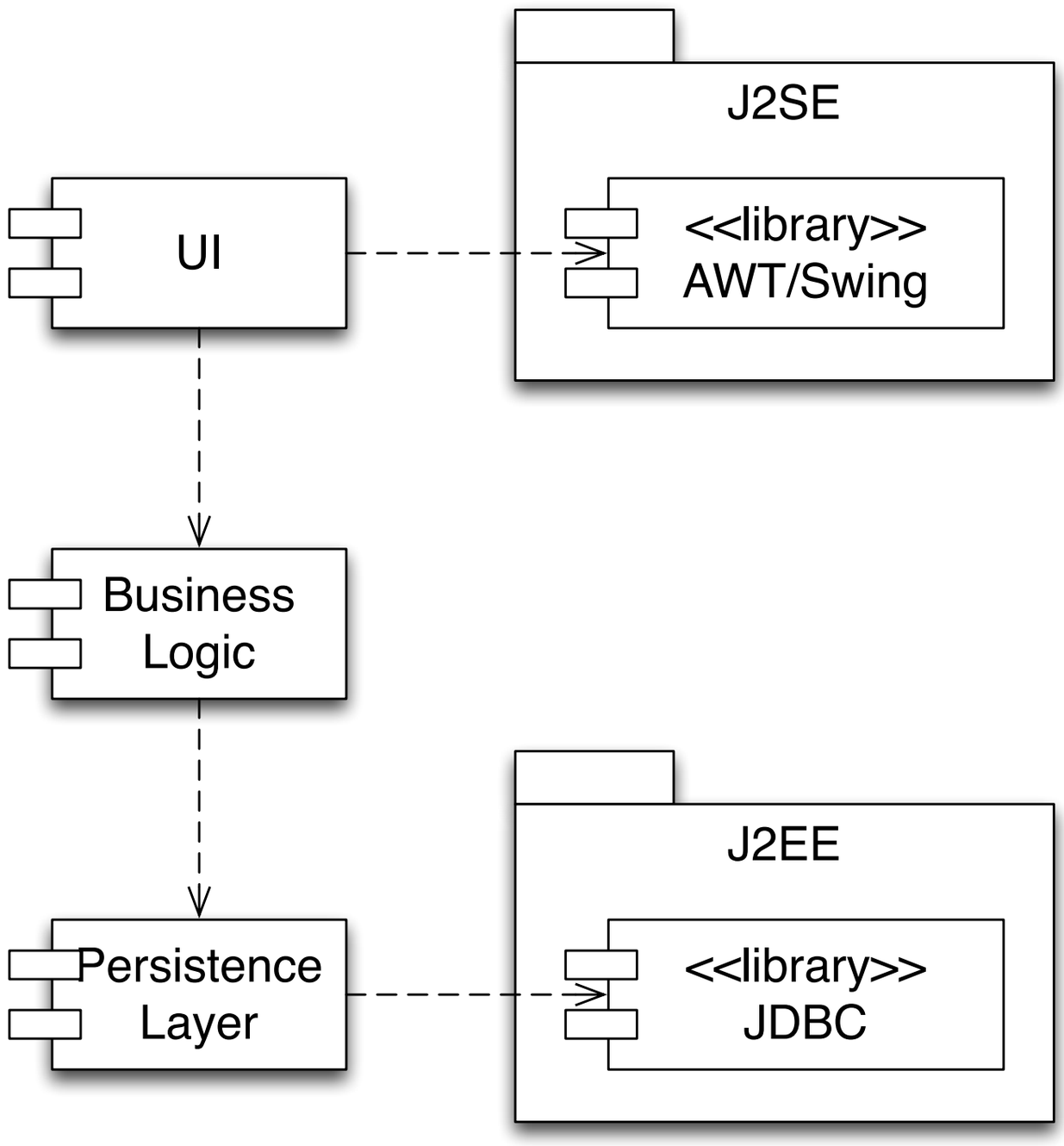
# Package diagram



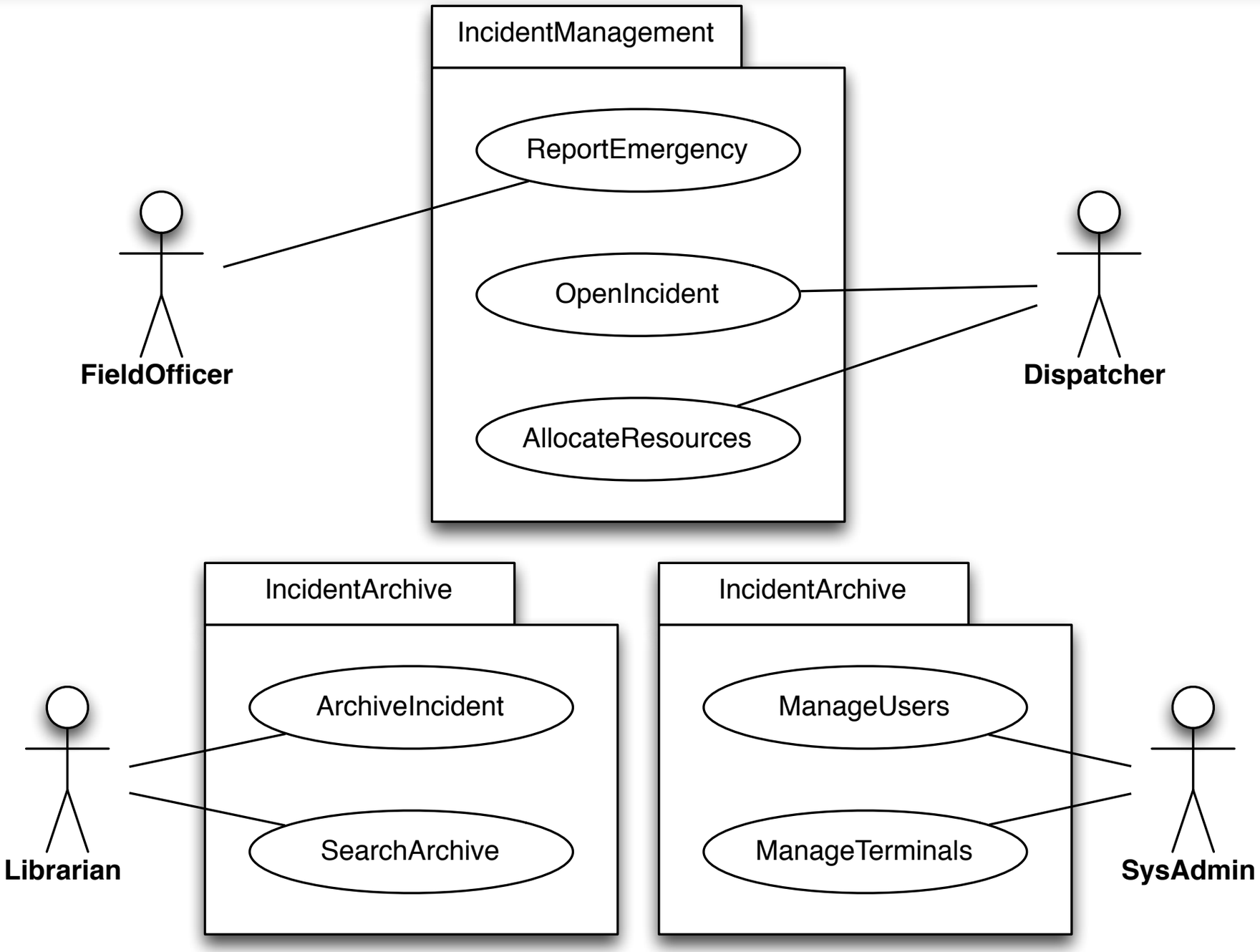

- Structure diagrams.
- Very versatile.
- They are used to depict how various packages of a system are connected.


Web Shopping
Mobile Shopping
Phone Shopping
Mail Shopping
«merge»
«merge»
package
«use»
«use»
package merge
«use»
Payment
«access»
Shopping Cart
usage dependency
private import
«import»
«import»
uml-diagrams.org
public import
Customer
Inventory
package

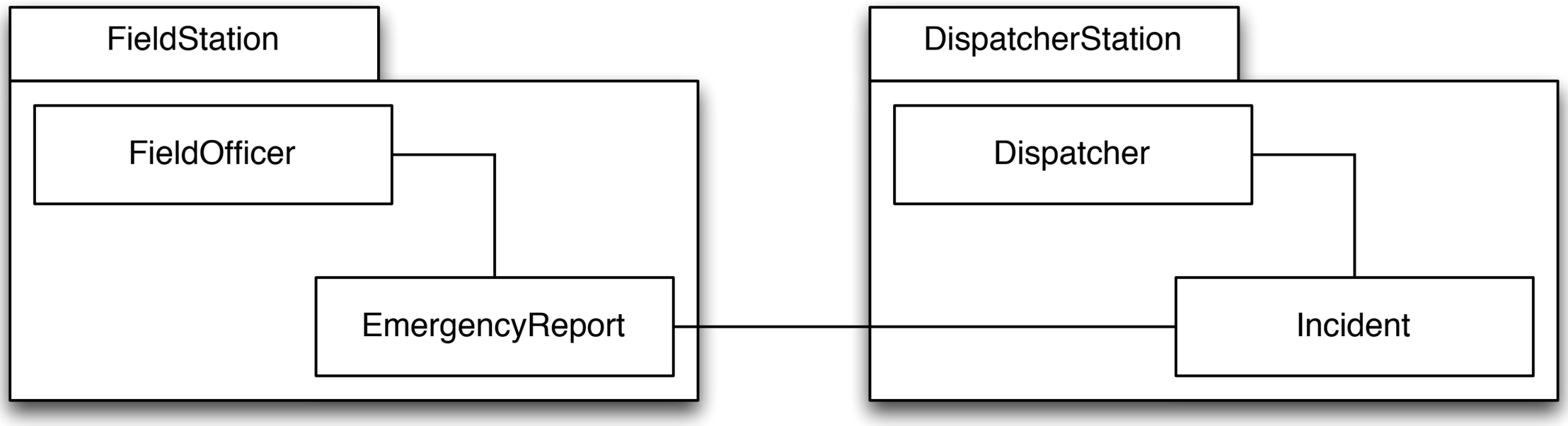
# Package and component



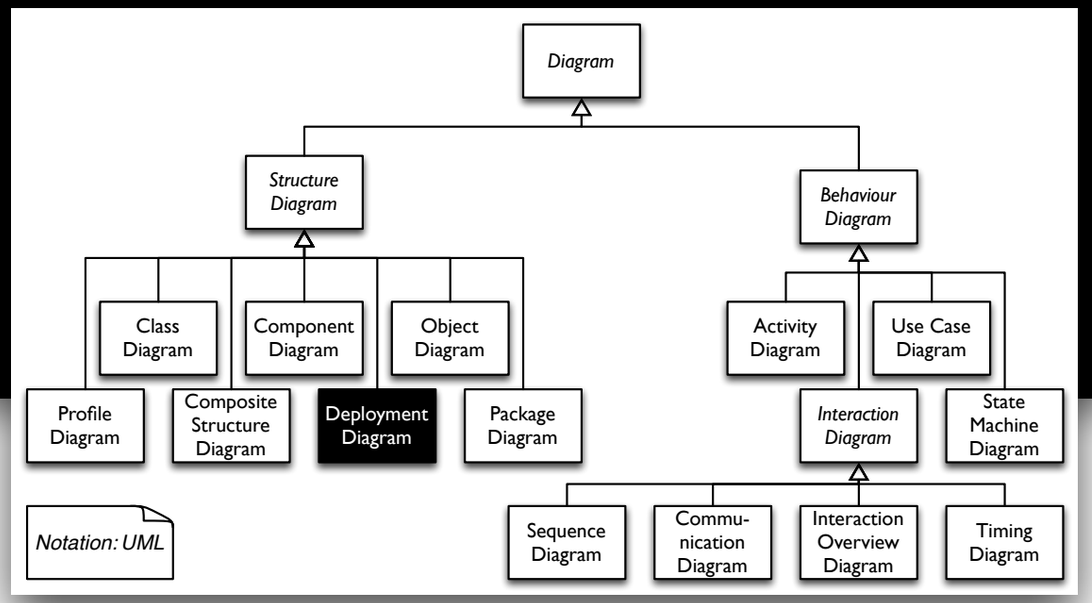
# Package and use case



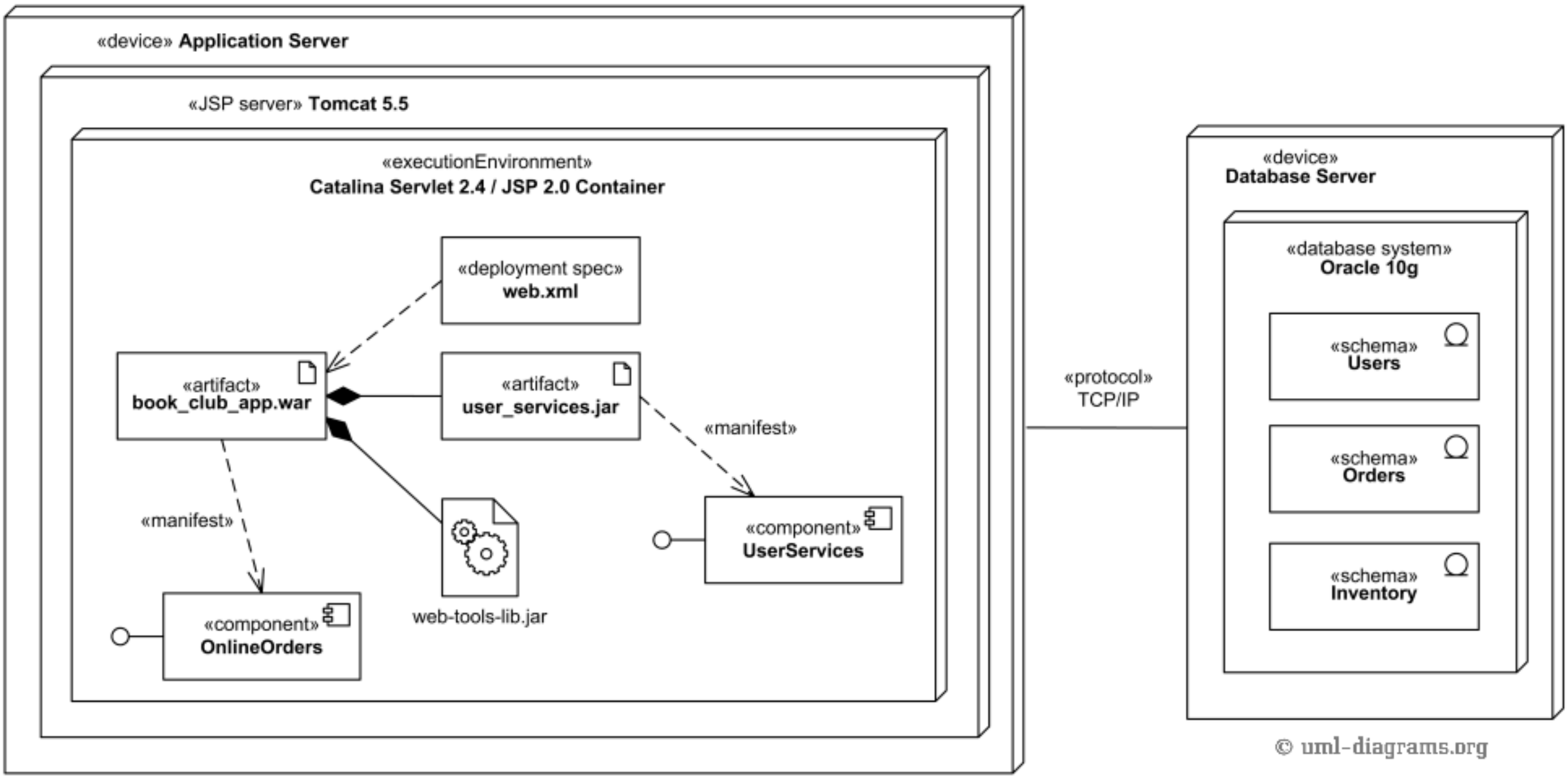
# Package and class

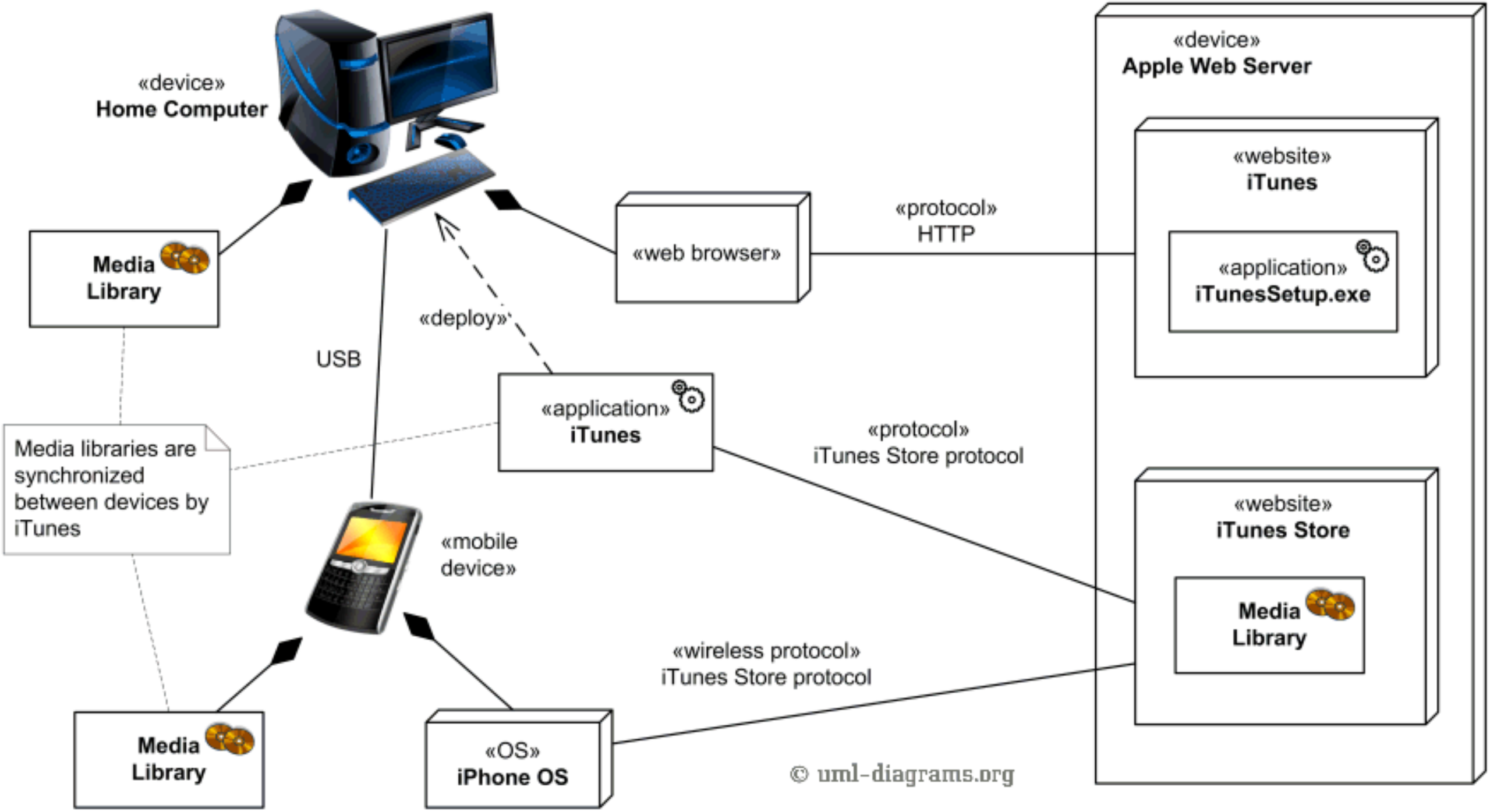


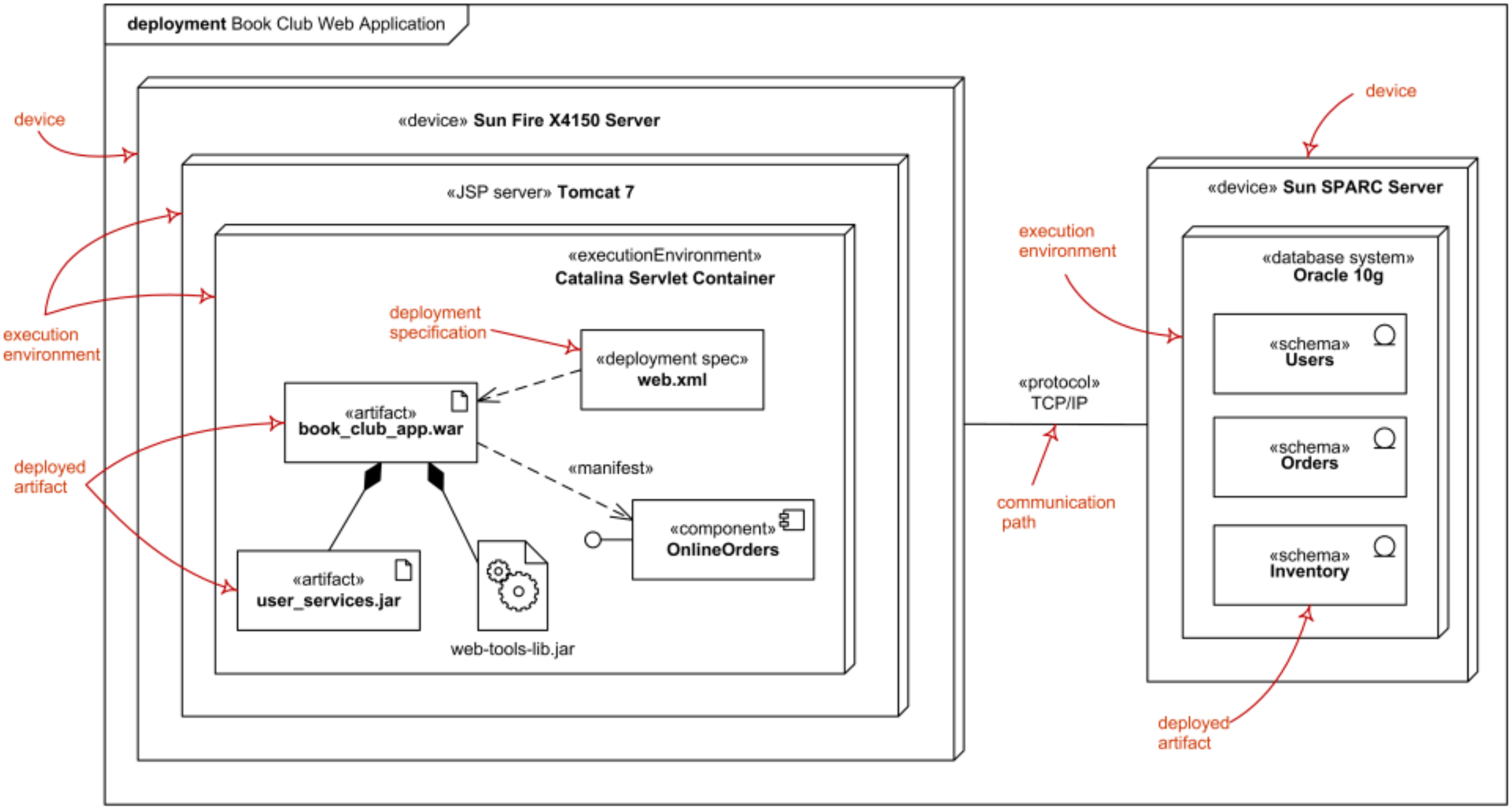
# Deployment diagram

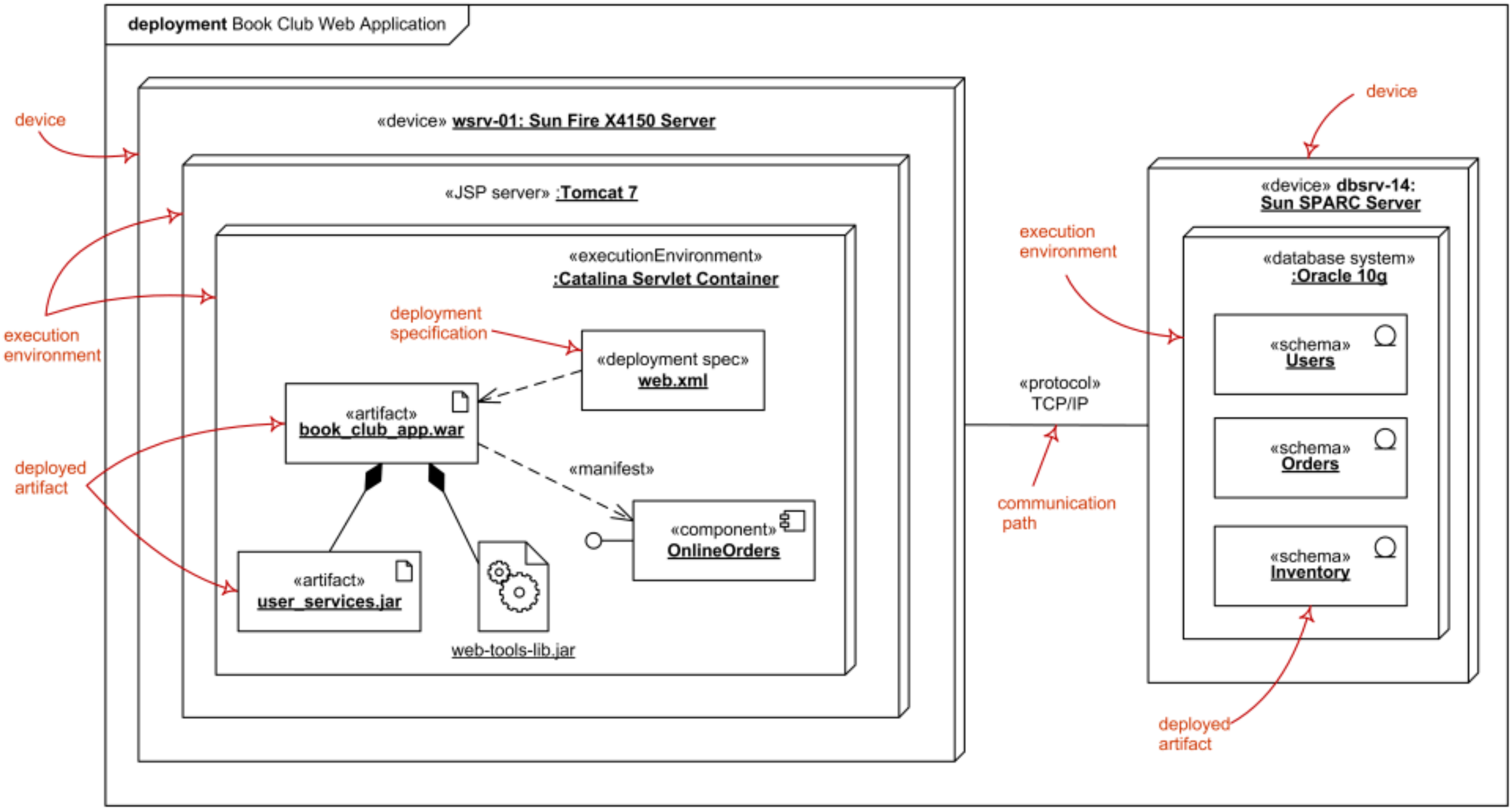


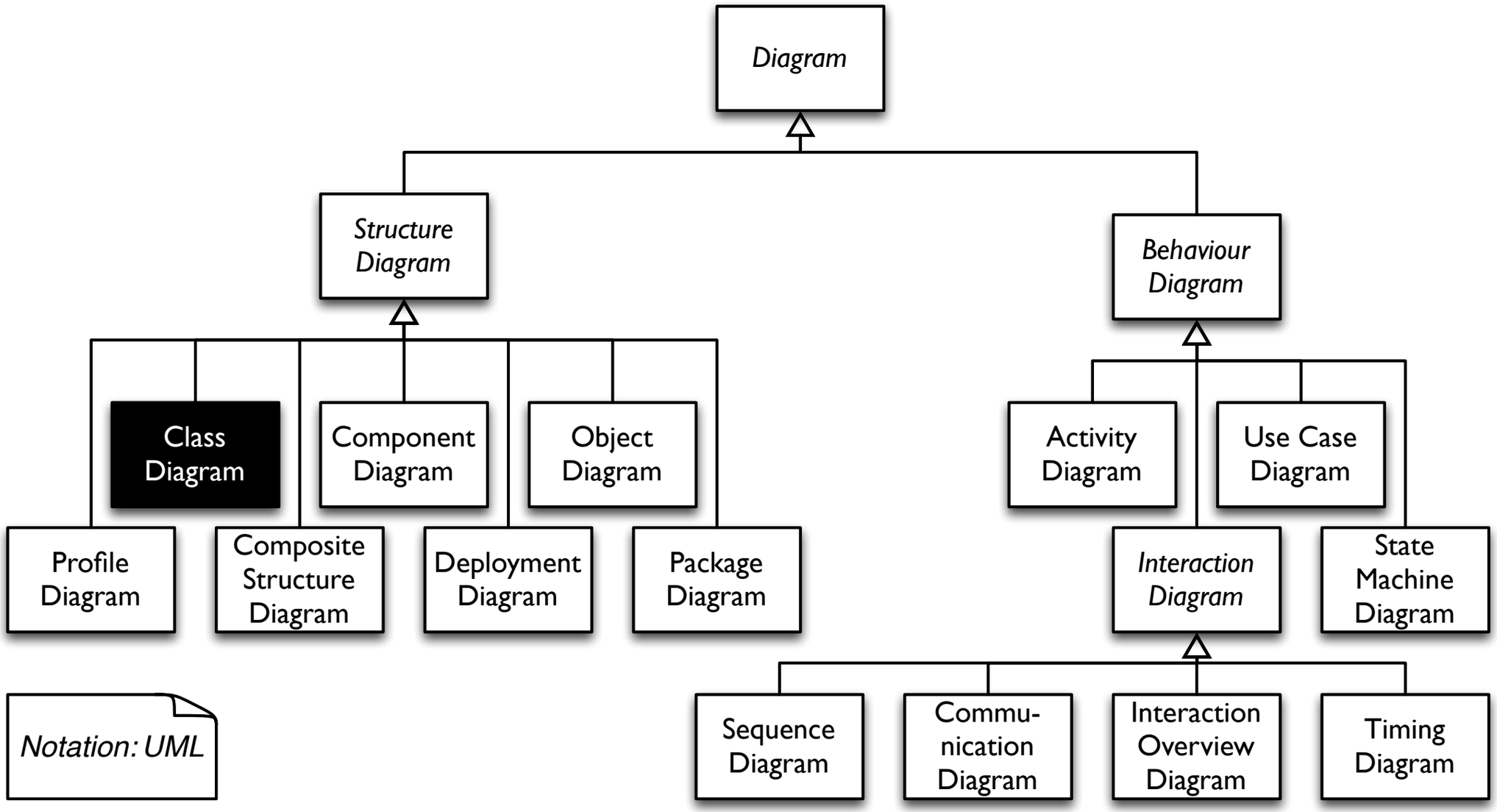

- Structure diagrams.
- They are used to depict where the various elements of a system are located.
- For instance, a distributed system based on a client/server architecture.
- Deployment diagrams contain
  - nodes and communication paths
  - components and dependancies


«device» Application Server
«JSP server» Tomcat 5.5
«executionEnvironment»
Catalina Servlet 2.4 / JSP 2.0 Container
«deployment spec»
web.xml
«artifact»
book_club_app.war
«artifact»
user_services.jar
«manifest»
«manifest»
«component»
UserServices
web-tools-lib.jar
«component»
OnlineOrders
«protocol»
TCP/IP
«device»
Database Server
«database system»
Oracle 10g
«schema»
Users
«schema»
Orders
«schema»
Inventory
© uml-diagrams.org


«device»
Home Computer
«device»
Apple Web Server
«website»
iTunes
«application»
iTunesSetup.exe
Media
Library
«web browser»
«protocol»
HTTP
USB
«deploy»
«application»
iTunes
Media libraries are synchronized between devices by iTunes
«protocol»
iTunes Store protocol
«website»
iTunes Store
Media
Library
«mobile
device»
«wireless protocol»
iTunes Store protocol
Media
Library
«OS»
iPhone OS
© uml-diagrams.org


deployment Book Club Web Application
device
«device» Sun Fire X4150 Server
«JSP server» Tomcat 7
«executionEnvironment»
Catalina Servlet Container
execution environment
deployment specification
«deployment spec»
web.xml
«artifact»
book_club_app.war
deployed artifact
«manifest»
«component»
OnlineOrders
«artifact»
user_services.jar
web-tools-lib.jar
device
«device» Sun SPARC Server
execution environment
«database system»
Oracle 10g
«schema»
Users
«schema»
Orders
«schema»
Inventory
«protocol»
TCP/IP
communication path
deployed artifact


deployment Book Club Web Application
device
«device» wsrv-01: Sun Fire X4150 Server
«JSP server» :Tomcat 7
execution environment
«executionEnvironment» :Catalina Servlet Container
deployment specification
«deployment spec» web.xml
«artifact» book_club_app.war
deployed artifact
«manifest»
«component» OnlineOrders
«artifact» user_services.jar
web-tools-lib.jar
«protocol» TCP/IP
communication path
device
«device» dbsrv-14: Sun SPARC Server
execution environment
«database system» :Oracle 10g
«schema» Users
«schema» Orders
«schema» Inventory
deployed artifact




# Diving into class diagrams

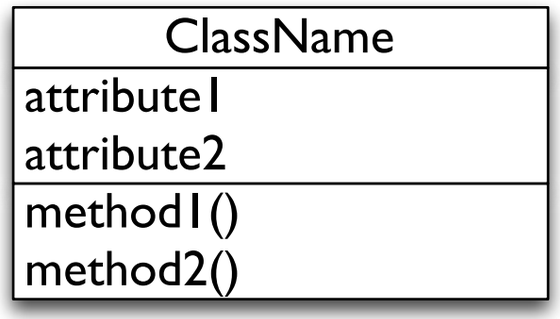
# The class symbol

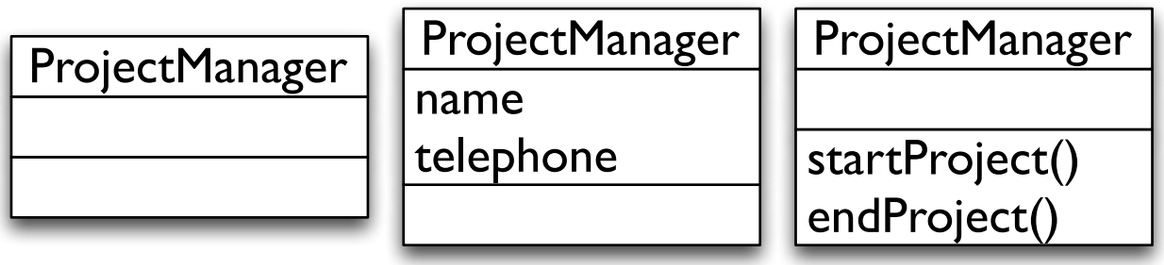
- Areas
  - Class name
  - Attributes
  - Methods (signatures)

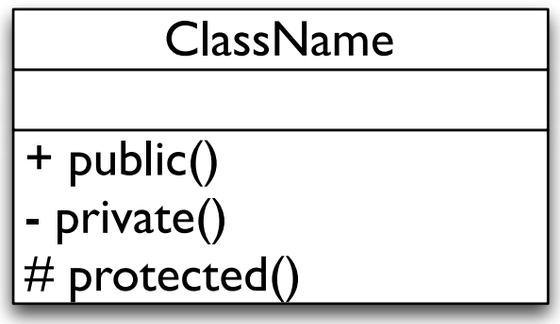
- Abbreviations
  - Without attributes and methods
  - Without attributes or methods

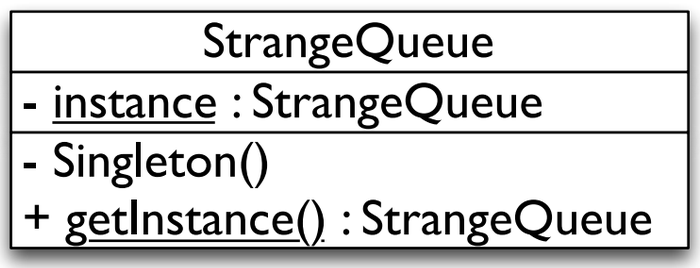
- Visibility
  - Public
  - Private
  - Protected
  - Internal (~)
  - Protected Internal (#~)

- Underline when static



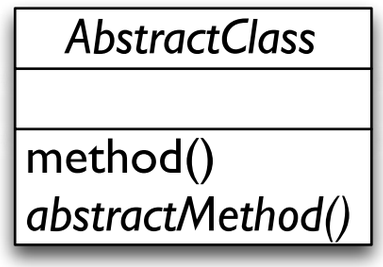



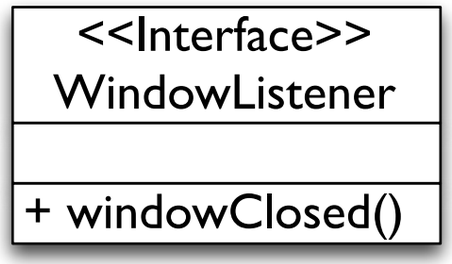



# Abstract classes and interfaces

- Abstract classes
  - Italic
  - or using the label {abstract}
  - holds also for methods




- Interfaces
  - Same as the classes
  - Use the stereotype <<Interface>>
  - or the lollipop notation

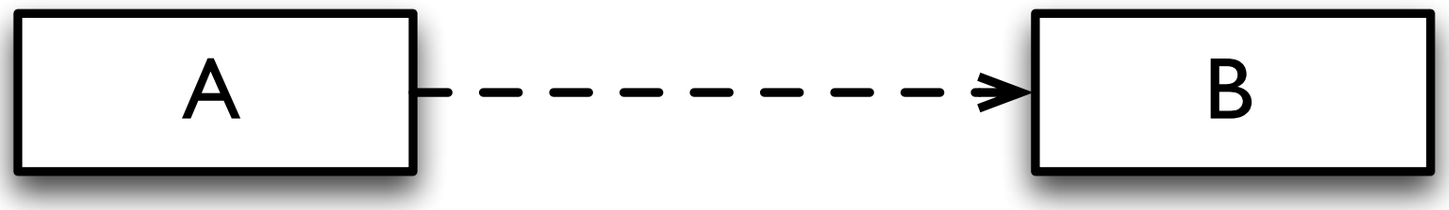



# Relationships between classes

- Relationships between classes (and interfaces and packages):
  - Dependancy
  - Association
  - Aggregation/Composition
  - Inheritance

- Navigability, Multiplicity, Roles, and Qualification.

# Relationships: generic dependancy

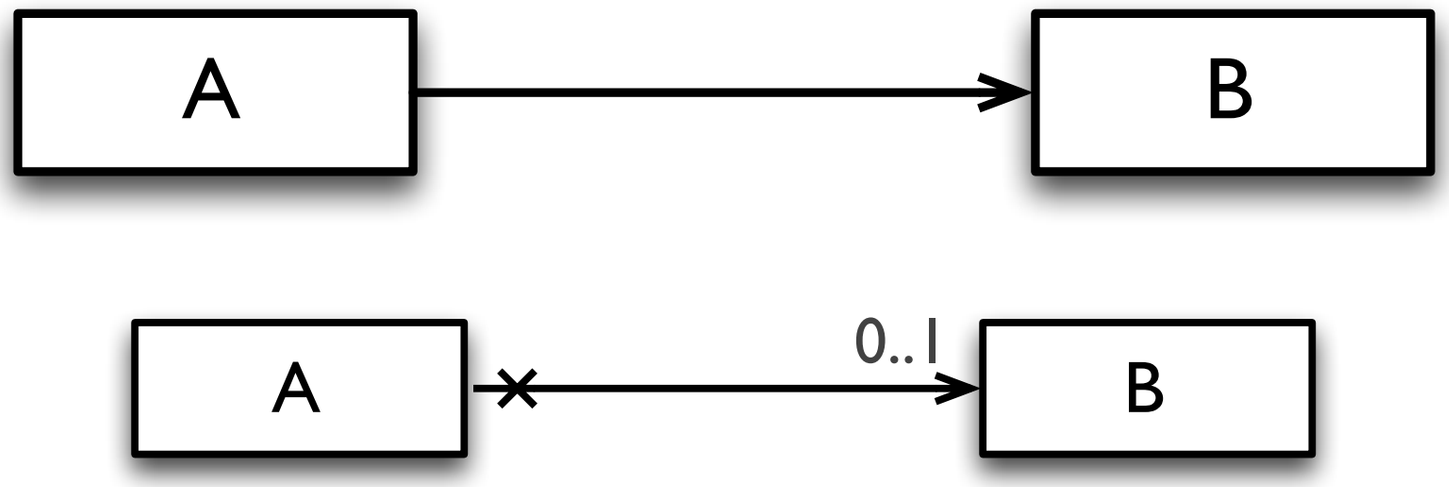



- A depends on B if A needs B to execute.
  - A calls B
  - A uses the type B

# Relationships: generic association


- A generic (but interesting) relationship between A and B.

# Navigability




- Arrow that indicates if the instance of A has direct access to the instance of B.

# Multiplicity and roles

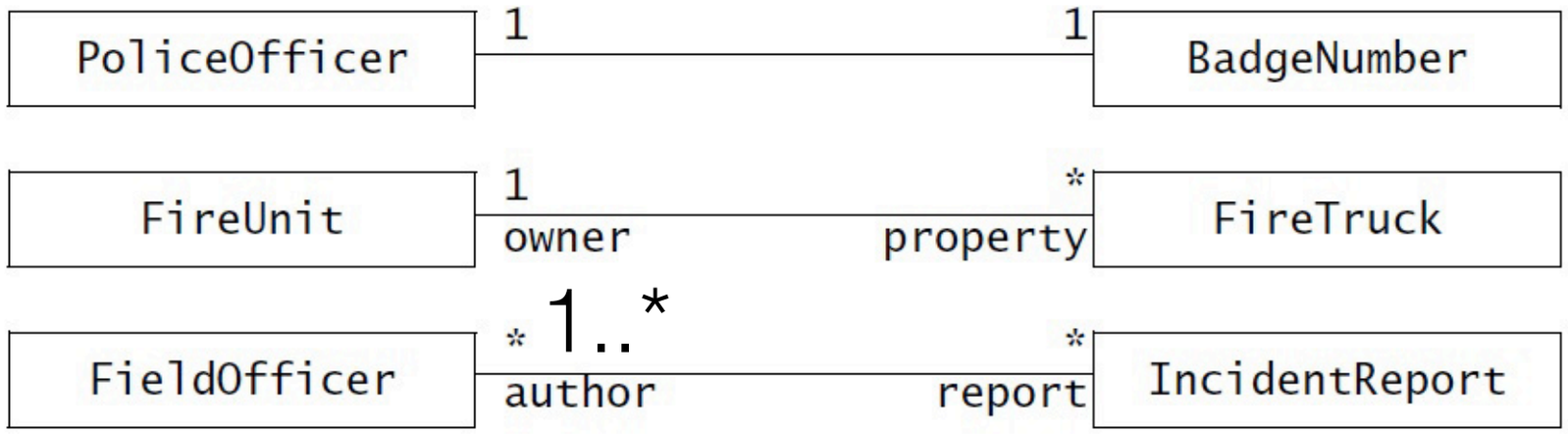
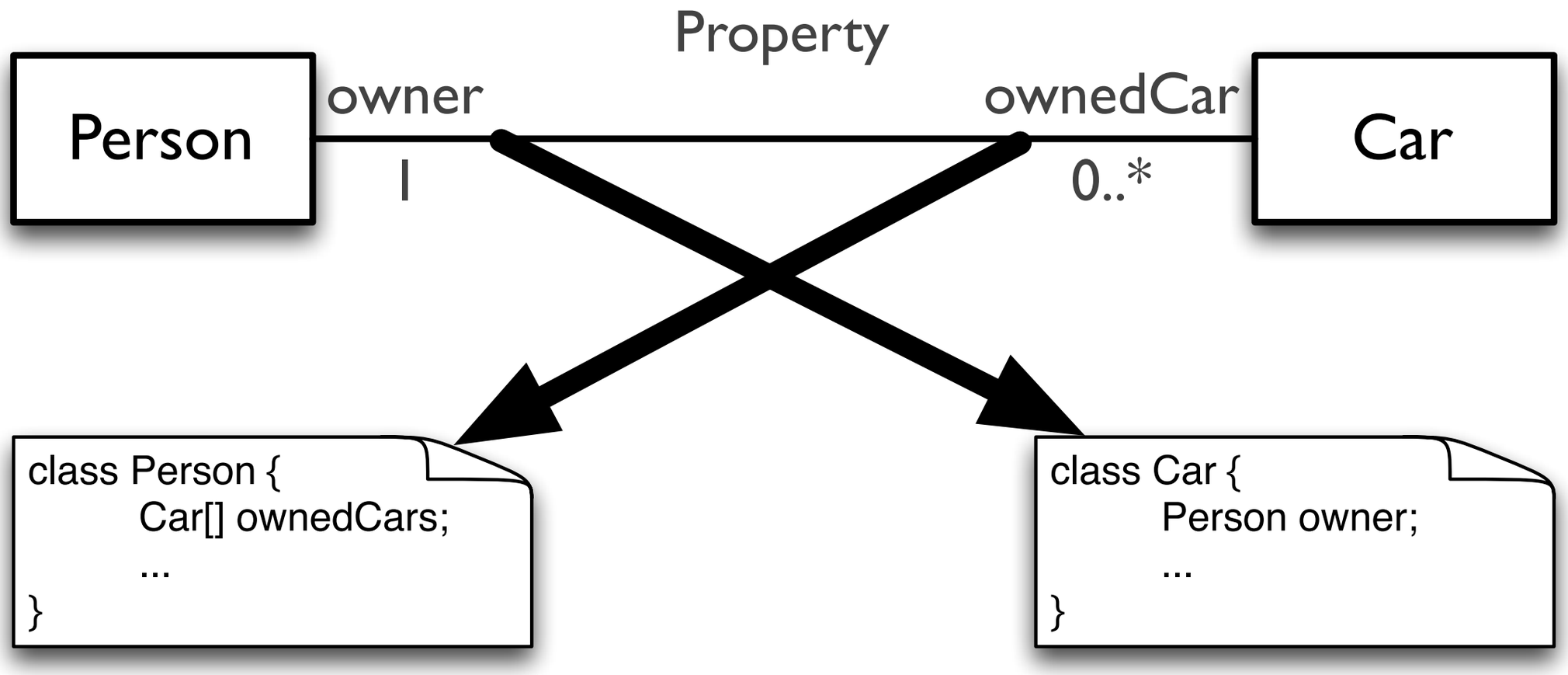



**Figure 2-27** Examples of multiplicity (UML class diagram). The association between `PoliceOfficer` and `BadgeNumber` is one-to-one. The association between `FireUnit` and `FireTruck` is one-to-many. The association between `FieldOfficer` and `IncidentReport` is many-to-many.



- Indicates how many instances for each relationship are allowed.

# An example


Property
Person
owner
1
ownedCar
0..*
Car
class Person {
Car[] ownedCars;
...
}
class Car {
Person owner;
...
}

# Qualification

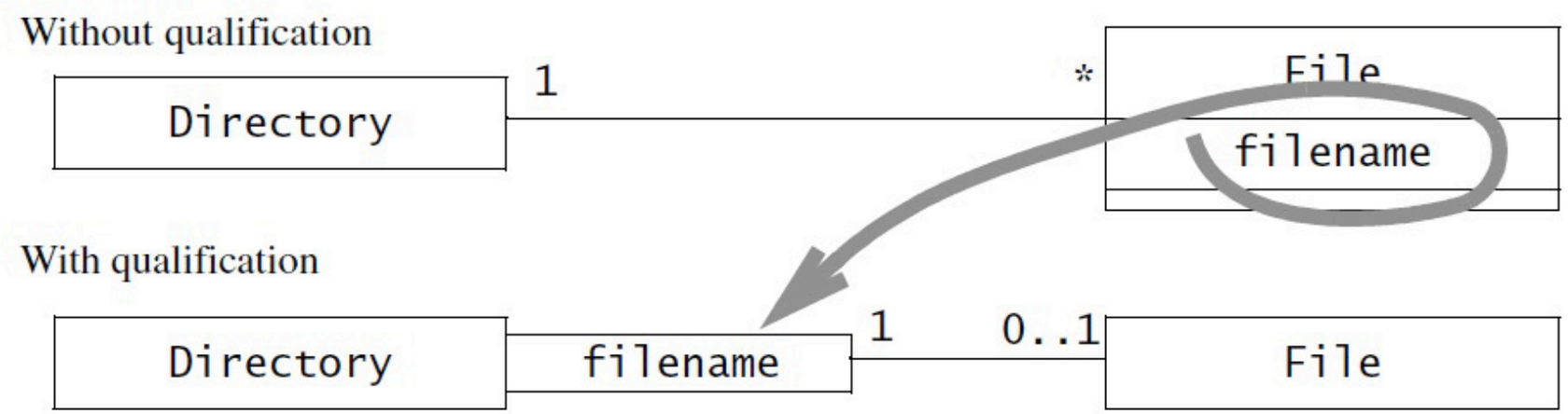
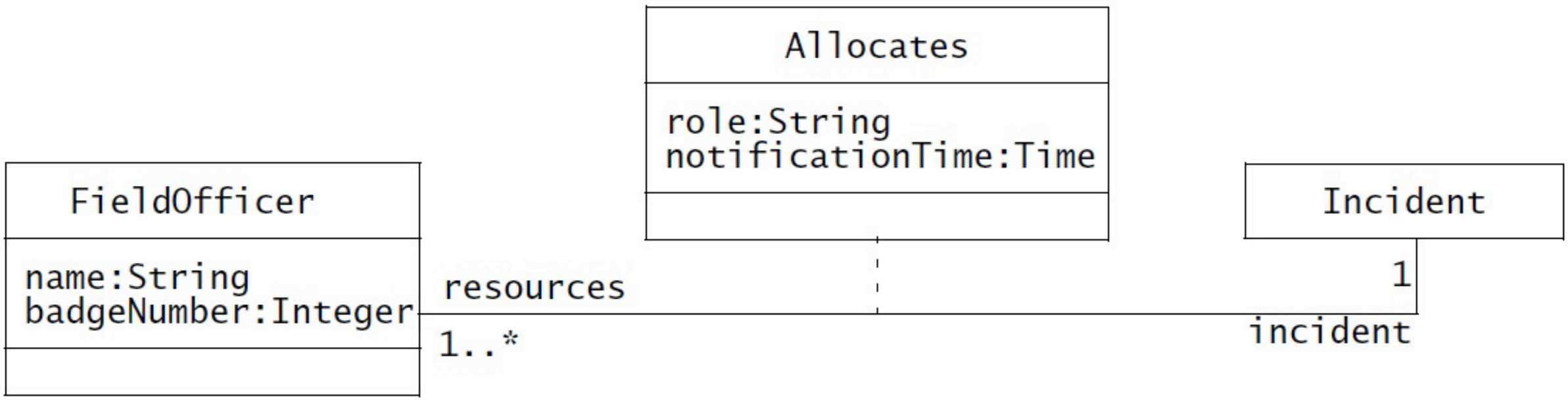


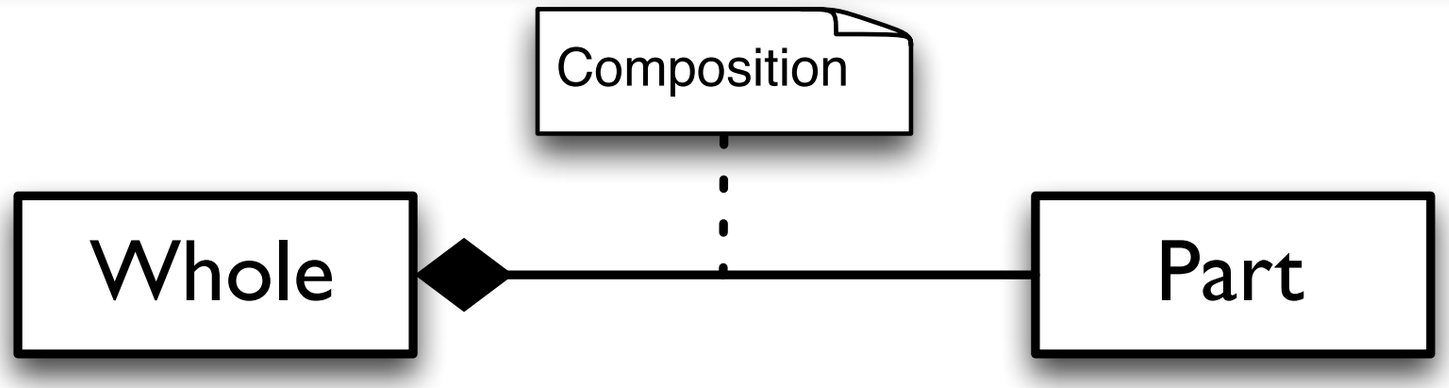

**Figure 2-31** Example of how a qualified association reduces multiplicity (UML class diagram). Adding a qualifier clarifies the class diagram and increases the conveyed information. In this case, the model including the qualification denotes that the name of a file is unique within a directory.

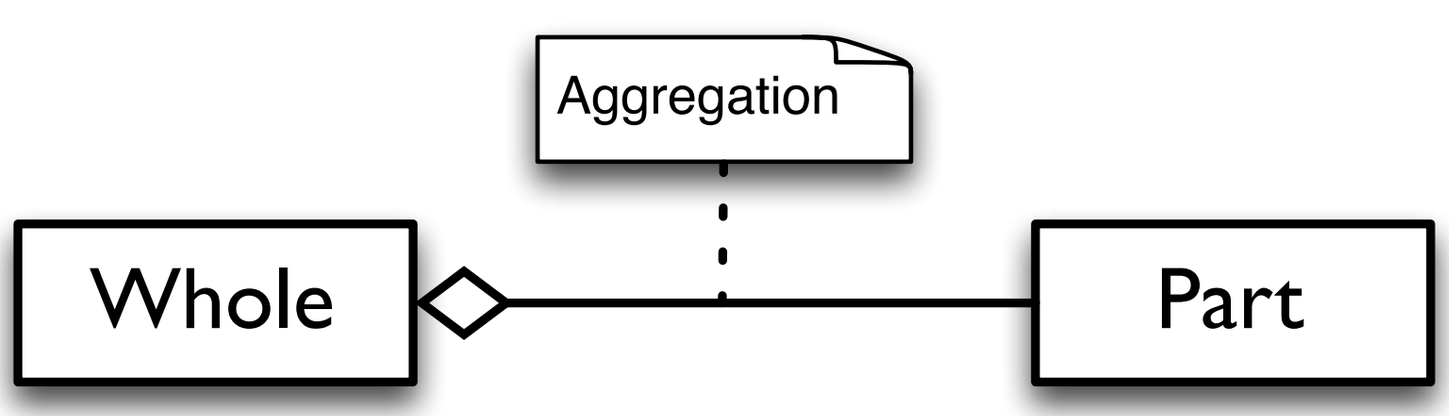
# Relationships: association class




- An association may be refined to have its own set of features; that is, features that do not belong to any of the connected classifiers but rather to the association itself.

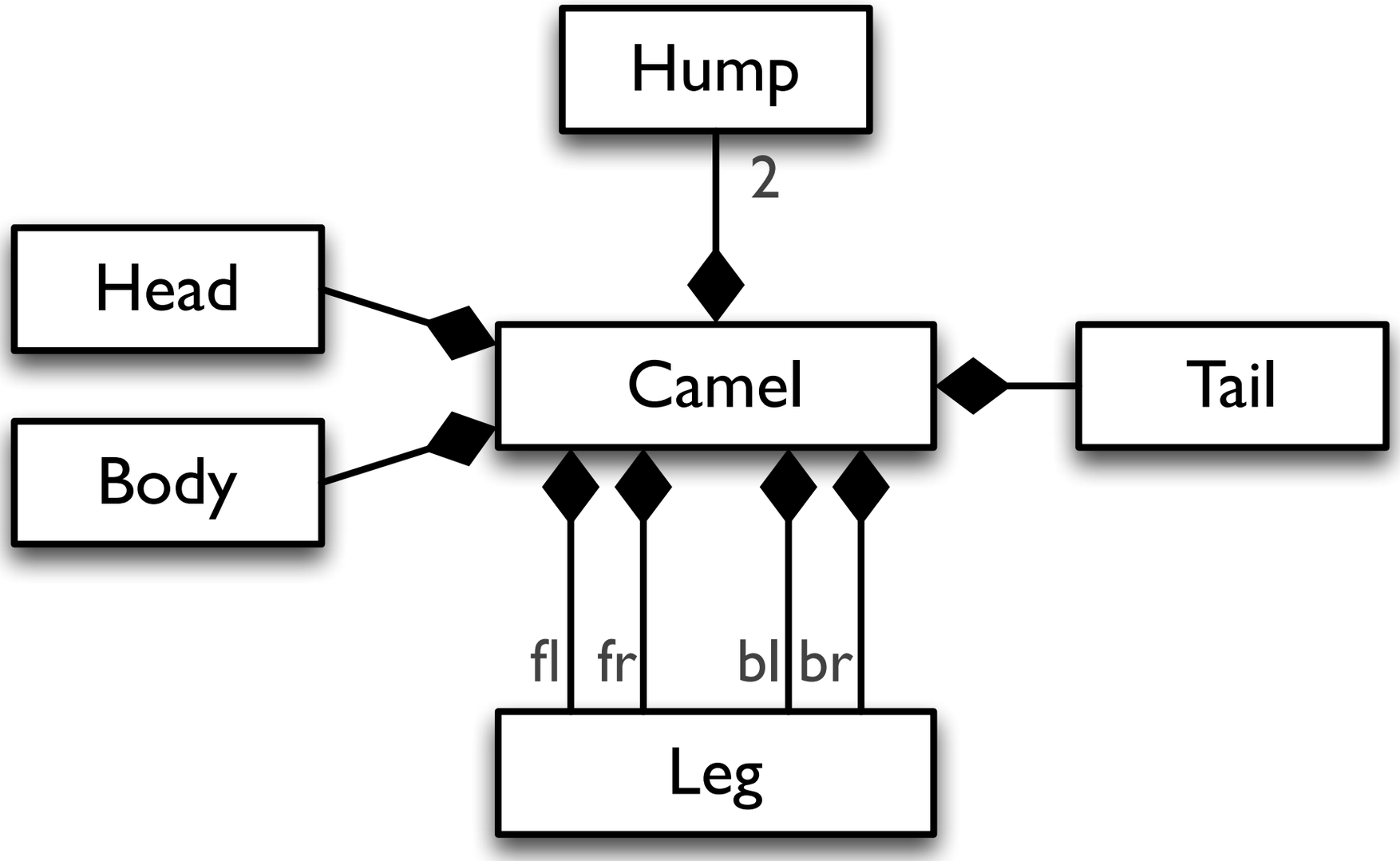
# Relationships: aggregation/composition




- They represent a whole/part relationship between classes.
- Composition is stronger than aggregation. In the composition relationship the whole cannot exist without the parts and "usually" when the composite (whole) is deleted, so are the components (parts).

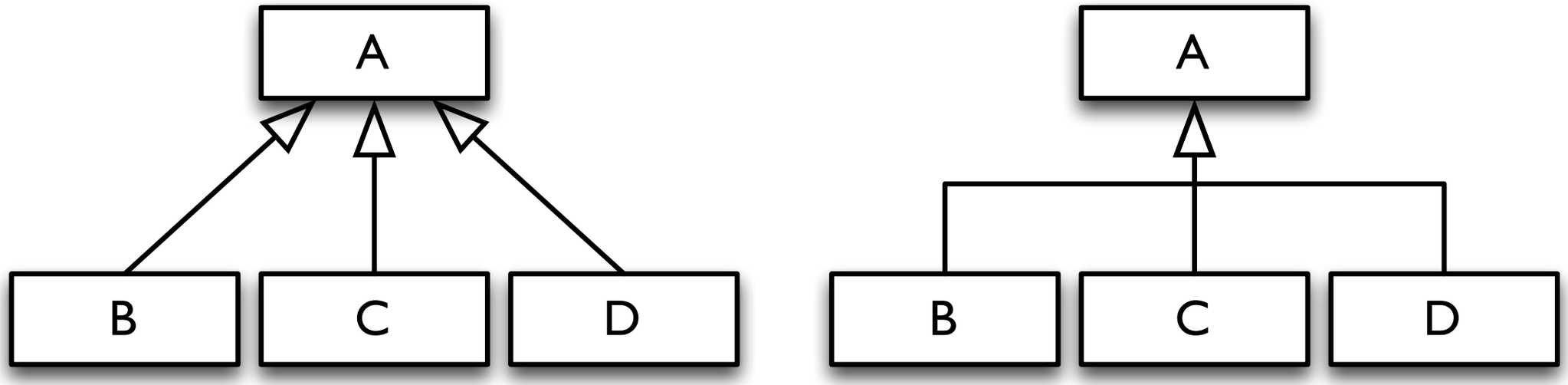
# An example of composition



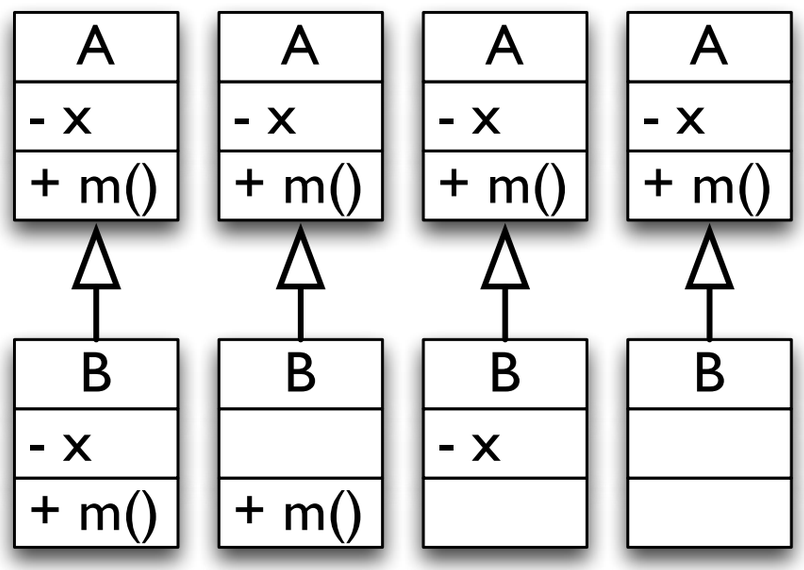
# Note!

- It is not always easy to define whether a relationship is a composition or an aggregation …
- … it is important to know that a distinction exists,
- … it is important to know the implications of a composition over an aggregation,
- … however, in practice do not spend too much time on it!

# Relationships: inheritance

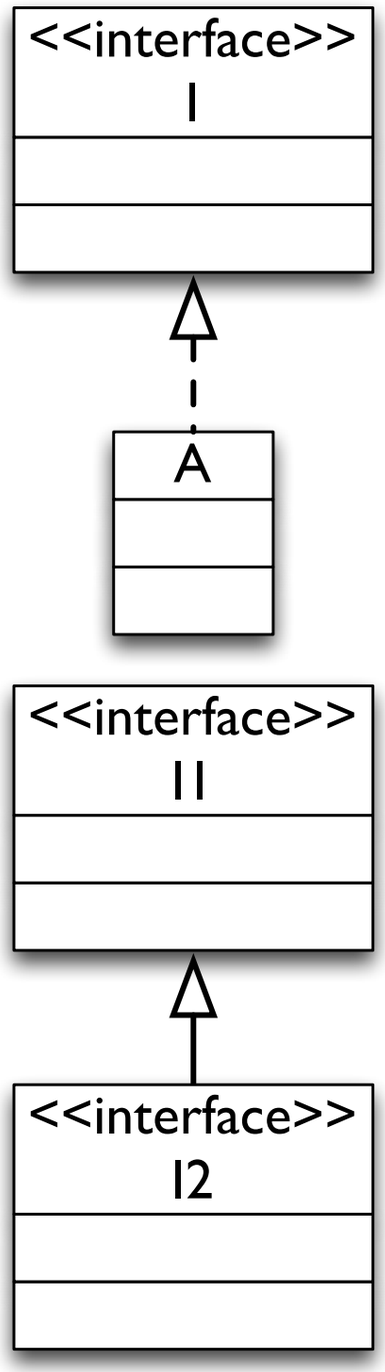


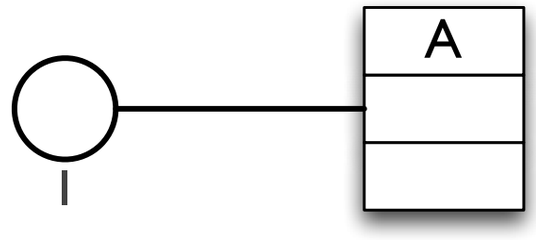

- B, C, D are a subclass of A.
  - B, C, D are a specialization of the superclass A.
  - B, C, D inherit from A.
- Attributes and methods inherited can be either shown or not (is it a relevant detail?).

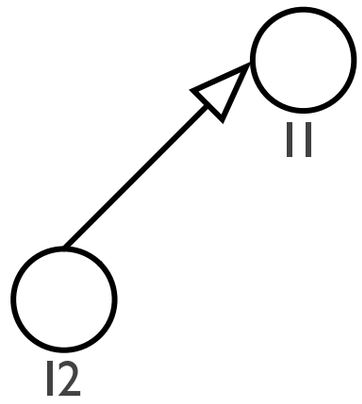


# Relationships: inheritance with interfaces

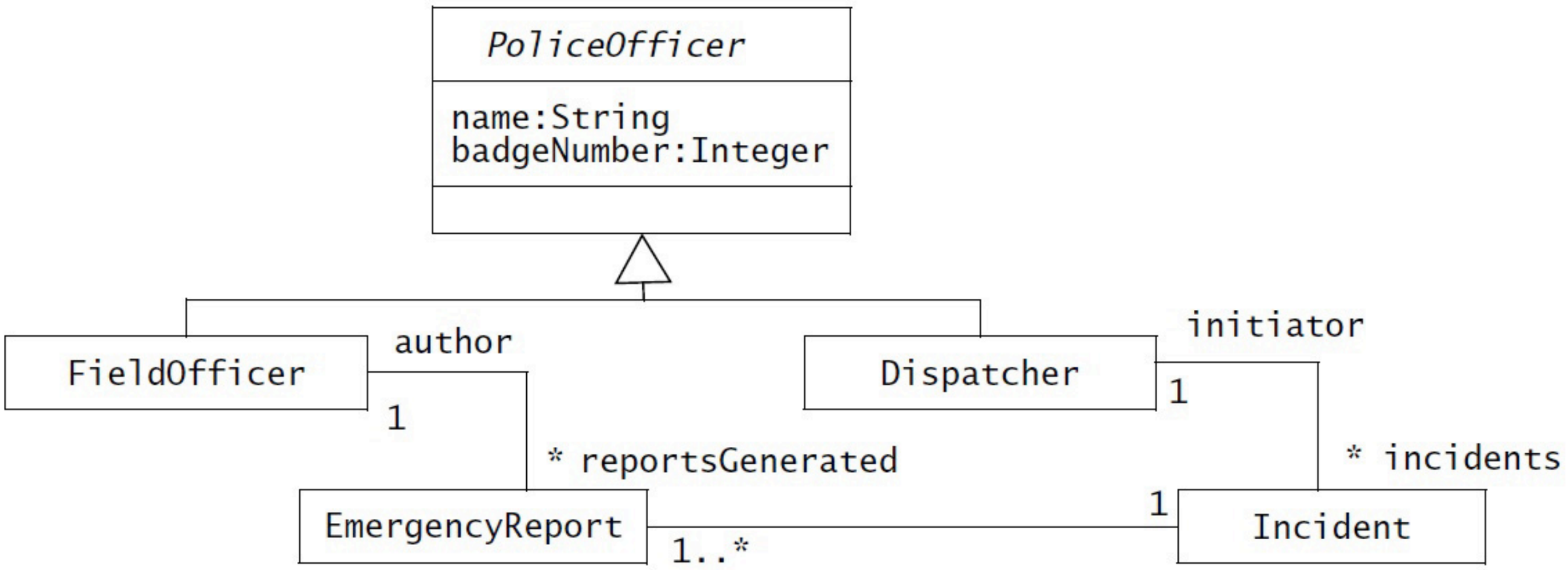
- A implements I
- I2 extends I1



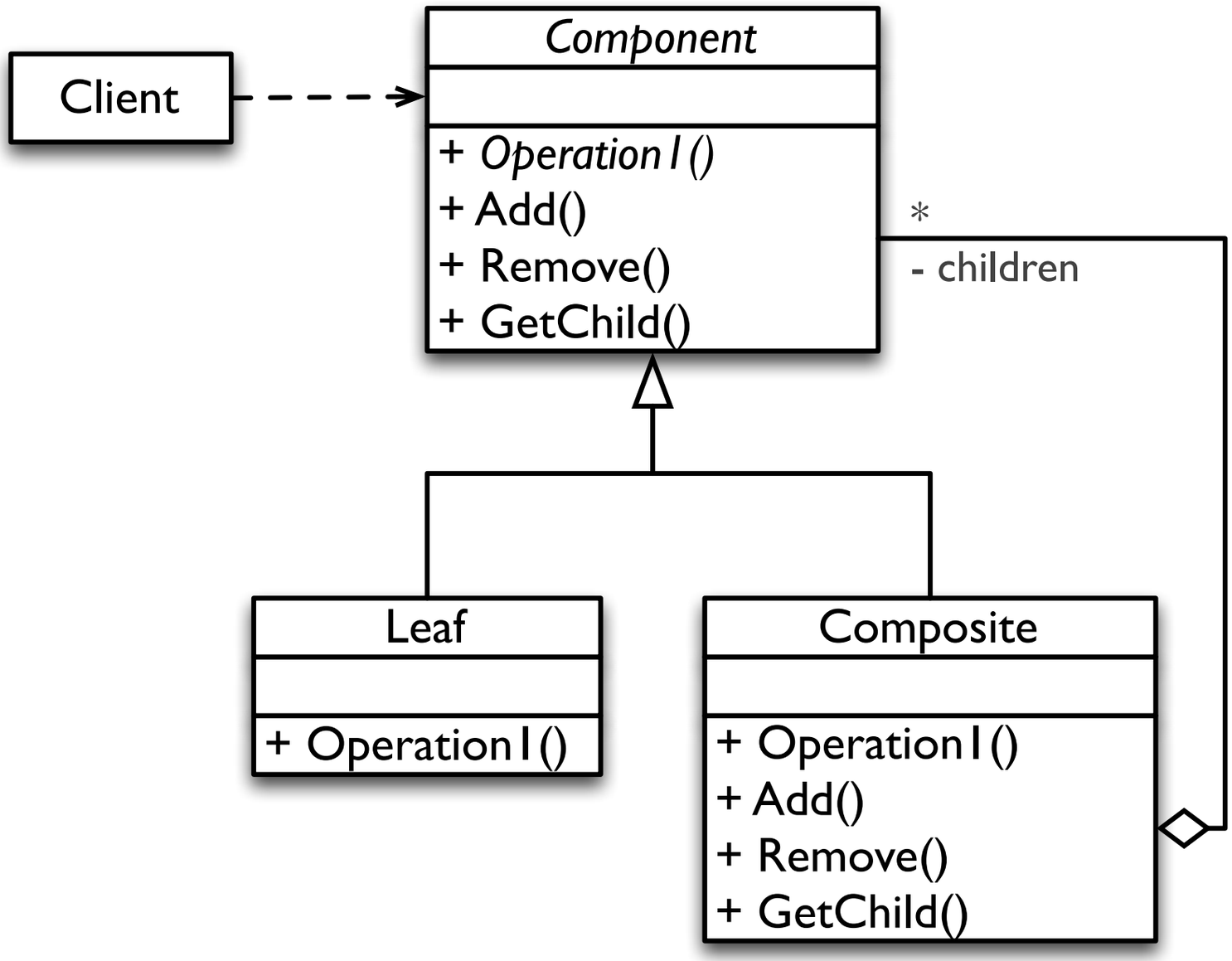



# An example

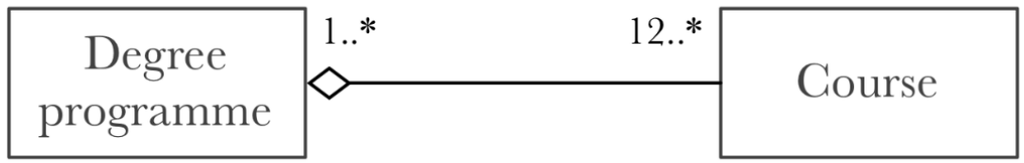

PoliceOfficer
name:String
badgeNumber:Integer
FieldOfficer
Dispatcher
author
1
initiator
1
* reportsGenerated
* incidents
EmergencyReport
1..*
1
Incident

# Another example



# From the questions pool

(5 points) Study the following diagram and answer the questions that follow it.




1. What kind of diagram is this?

   .....

2. Which type of UML relationship is this diagram an example of?

   .....

   .....

3. If a `Degree programme` object is destroyed, what happens to the `Course` objects that it contains?

   .....

   .....

4. How many `Course` objects are associated with each `Degree programme` object?

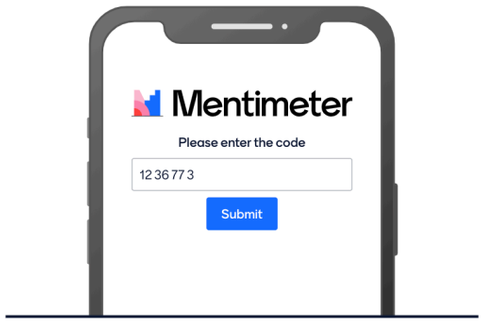
   .....

   .....

5. Can a `Course` object belong to more than one `Degree programme`?

Go to

www.menti.com




Enter the code

12 36 77 3

# Concluding

# What else is left about UML?

- Additional details about ALL the diagrams.
- Extensions.
- Object Constraint Language (OCL).
- Best practices.
- … but when necessary, we will cover additional topics as the necessity arises.

# Pareto principle - 80/20 rule

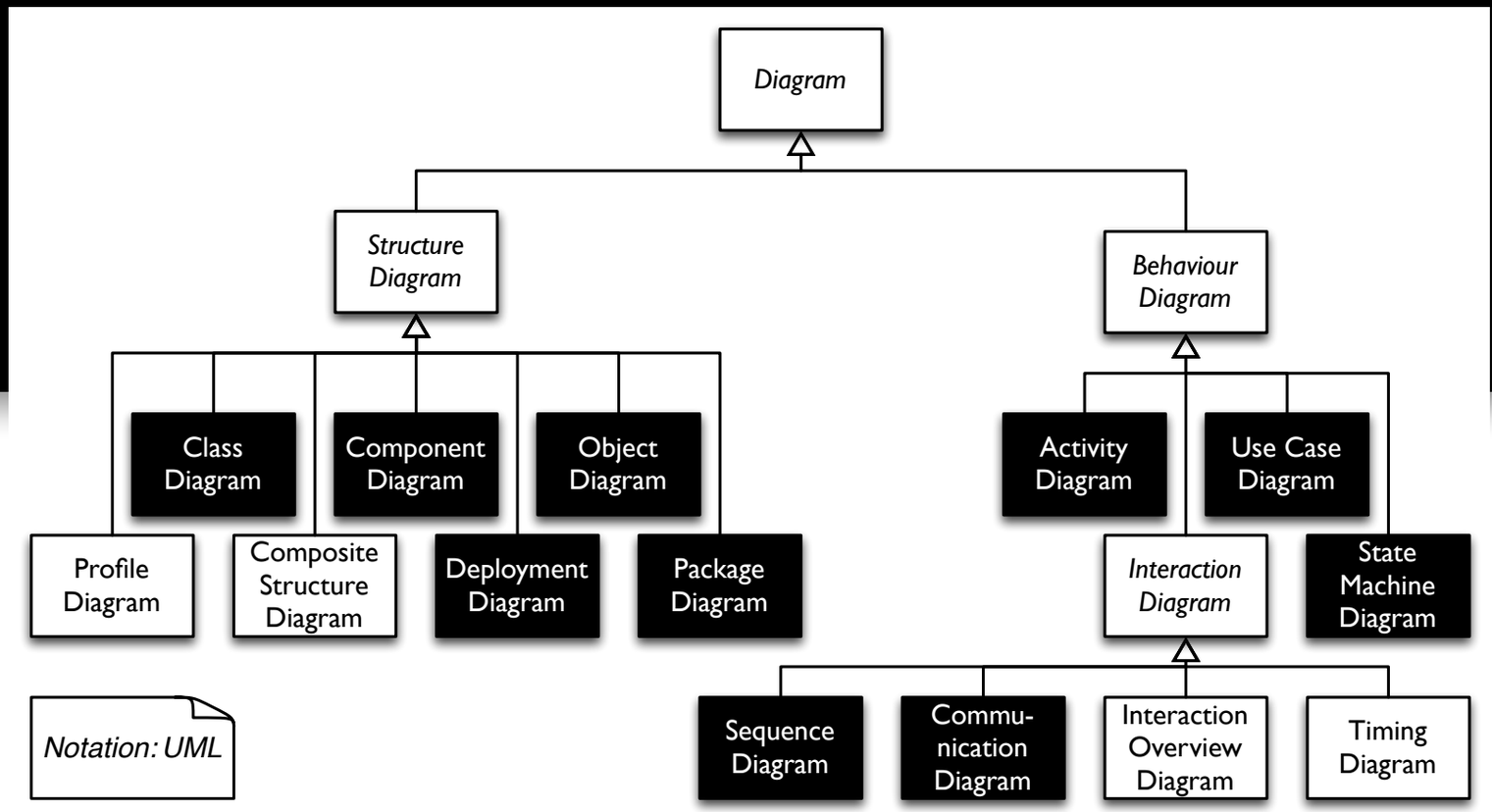
- [http://en.wikipedia.org/wiki/Pareto_principle]
  - 80% of your profits come from 20% of your customers
  - 80% of your complaints come from 20% of your customers
  - 80% of your profits come from 20% of the time you spend
  - 80% of your sales come from 20% of your products

- With UML is the same!
  - You can model 80% of most problems by using about 20% of the UML concepts.

# Key Points I

- System, modeling, model, view, notation.
- UML emerged victorious from the war of notations. An aggregation of several notation by the 3 amigos: Rumbaugh, Jacobson, Booch.
- 3 views on UML: as sketch, as blueprint, as a programming language.
- In this course you need to be able to draw syntactically correct UML diagrams – important part of the exam.
- Structure diagrams versus behaviour diagrams.

# Outline




- Literature
  - [OOSE] ch. 2
  - (Optional) [SE10] ch. 5

- Topics covered:
  - Modeling
  - UML and its history
  - UML diagrams
    - Brief description
    - Example(s)
    - Comment (or another example)
    - Template from [uml-diagrams.org]
  - A deeper dive into class diagrams