

# Object-Oriented Analysis and Design with UML

Adegboyega Ojo and Elsa Estevez

UNU-IIST

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# The Course: Objectives

- 1) present concepts of Object Oriented paradigm, as well as Object Oriented Analysis (OOA) and Design (OOD).
- 2) introduce the syntax, semantics, and pragmatics of UML and how to integrate it with the Unified Process
- 3) show how to articulate requirements using use cases
- 4) present how to develop structural and behavioural diagrams for the different views supported in UML
- 5) introduce concepts of Patterns and Frameworks and their applications in architecture and design tasks
- 6) present a comparative analysis of some major UML tools suitable industrial strength development

# The Course: Literature

---

- 1) OMG Unified Modeling Language Specification, Object Management Group.
- 2) UML Bible, Tom Pender, John Wiley and Sons, 2003.
- 3) Object-Oriented Analysis and Design using UML, Simon Bennet, Steve McRobb and Ray Farmer, McGraw-Hill, 2002.
- 4) Guide to Applying the UML, Sinan Si Alhir, Springer, 2002.
- 5) Object-Oriented Software Engineering, Bernd Bruegge, Allen H Dutoit, Prentice Hall, 2000.
- 6) The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999.
- 7) OO Software Development Using UML, UNU-IIST Tech Report 229.

# The Course: Approach

- 1) provide a general foundation of object orientation
- 2) discuss the basic building blocks of the UML – model elements, relationships, diagrams
- 3) present the UML diagrams as they relate to the core workflows of any development process:
  - a) **Requirements**: Use Case, Class, Object, Sequence and Statechart
  - b) **Architecture**: Collaborations and Components
  - c) **Design**: Class, Sequence, Statechart and Activity
  - d) **Implementation**: Components and Deployment
- 4) diagram features are presented in increasing details as required for each workflow

# Object Oriented Concepts

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# Overview: OO Concepts

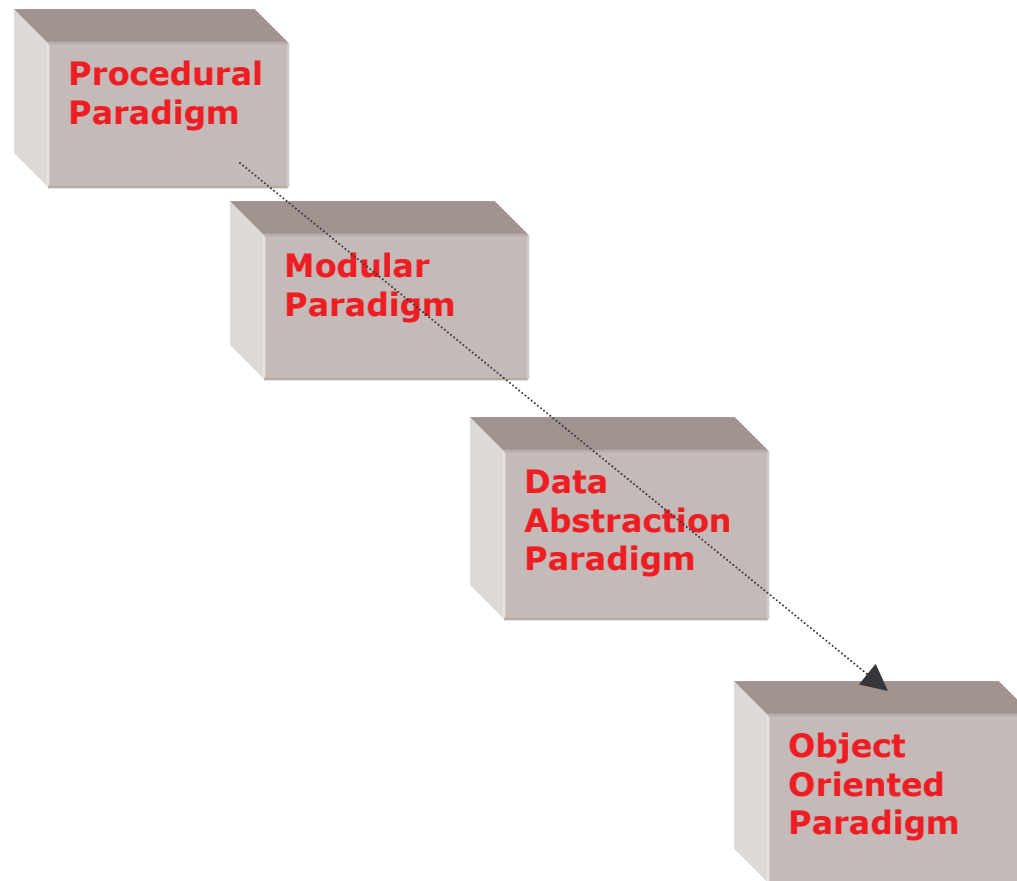
- 1) Background and Principles of Object Orientation
- 2) Object Oriented Concepts
- 3) Object Oriented Analysis (OOA)
- 4) Object Oriented Design (OOD)



# Background and Principles

# The Road to OO

---



# What is OO?

---

## Definition

Object orientation is about viewing and modelling the world (or any system) as a set of interacting and interrelated objects.

An approach characterized by the following features:

- 1) views the universe of discourse as consisting of interacting objects
- 2) describes and builds systems consisting of representation of objects

# Principles of OO

---

- 1) Abstraction
- 2) Encapsulation
- 3) Modularity
- 4) Hierarchy

# Abstraction

---

A model that includes most important aspects of a given system while ignoring less important details.

Abstraction allows us to manage complexity by concentrating on essential characteristics that makes an entity different from others.



Customer



Salesman

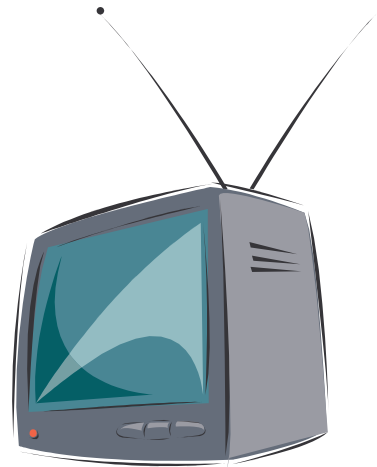
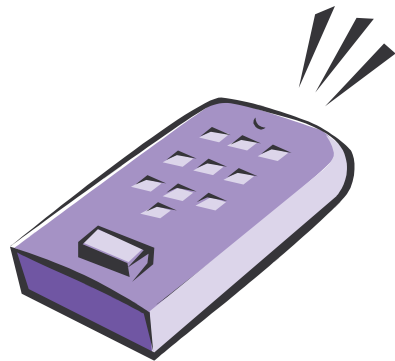


Product

*An example of an order processing abstraction*

# Encapsulation 1

- 1) Encapsulation separates implementation from users or clients.
- 2) Clients depend on interface.

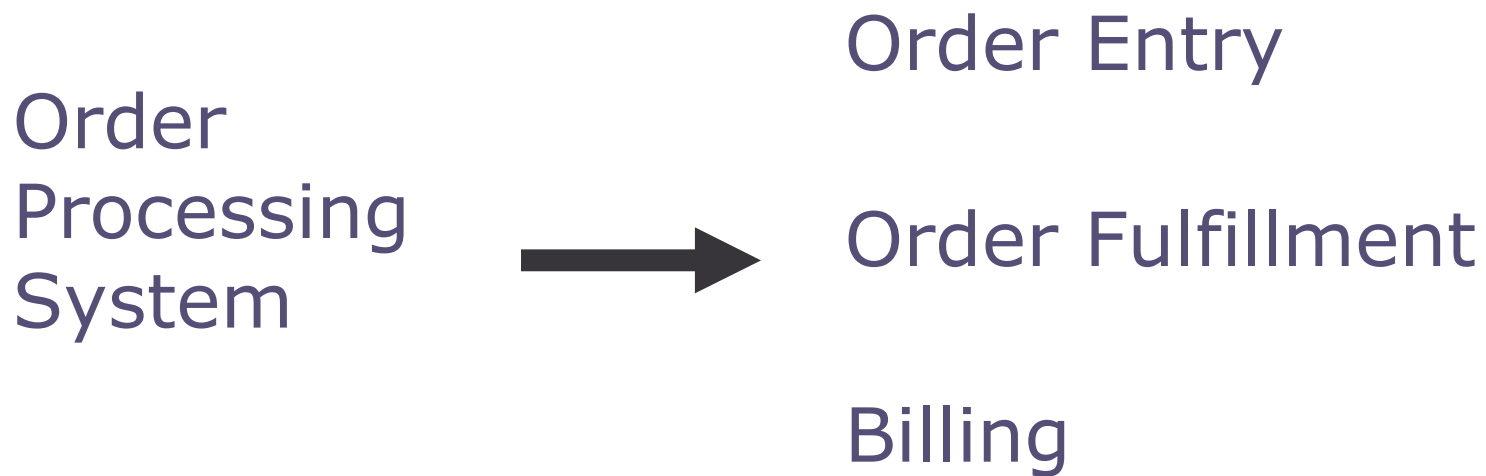


*Courtesy Rational Software*

# Modularity

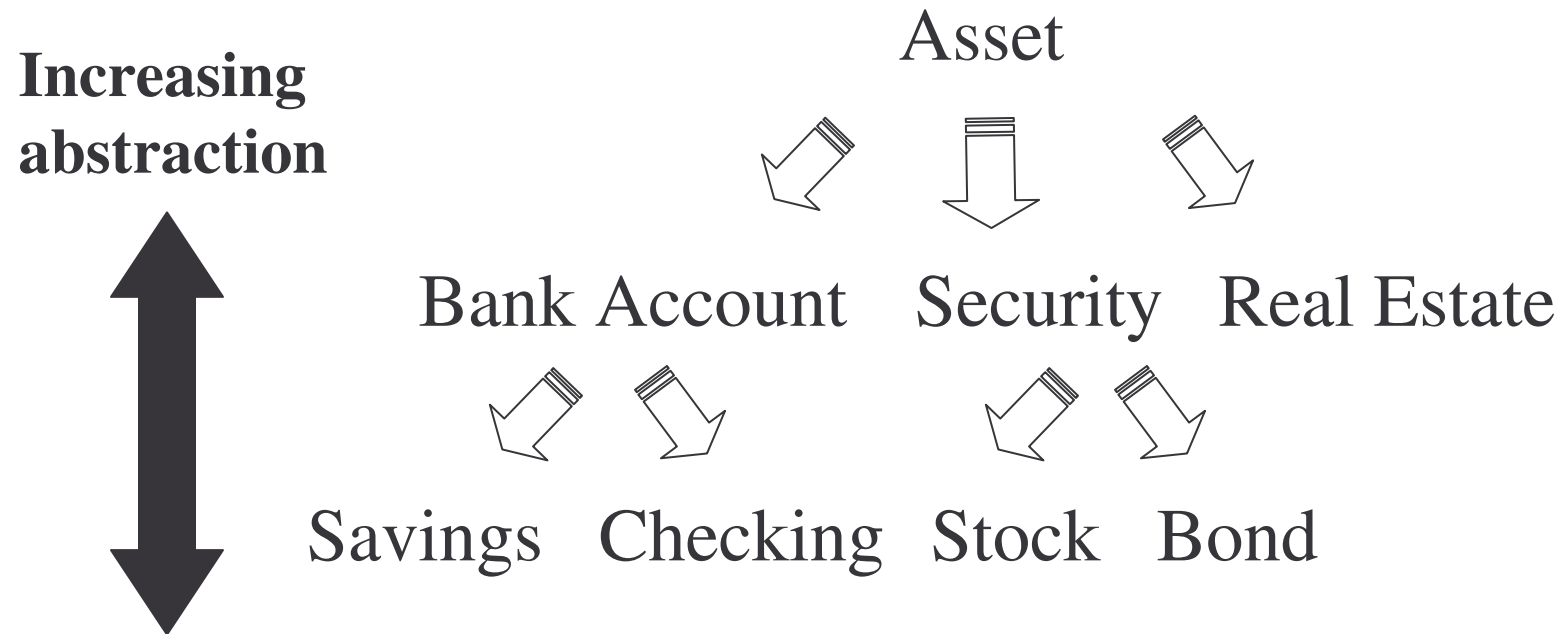
---

Modularity deals with the process of breaking up complex systems into small, self contained pieces that can be managed easily.



# Hierarchy

Is an ordering of abstractions into a tree like structure.





# Concepts

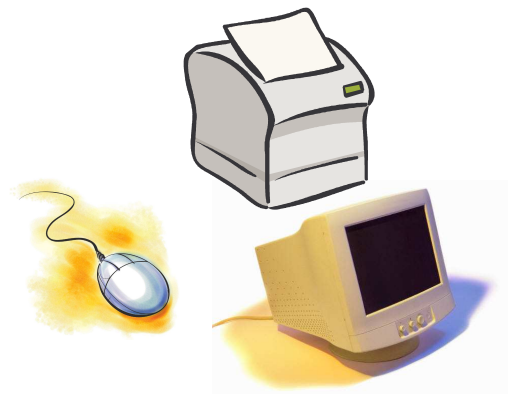
# Objects

---

What is an **object**?

- 1) any abstraction that models a single thing
- 2) a representation of a specific entity in the real world
- 3) may be tangible (physical entity) or intangible
- 4) examples: specific citizen, agency, job, location, order ...

# Example: Objects



# Object Definition

---

Two aspects: information and behaviour

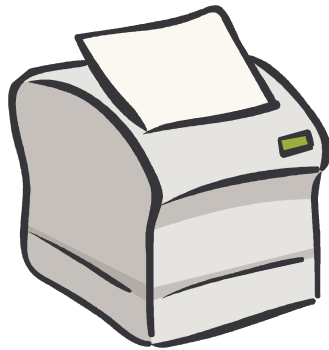
## Information:

- 1) has a unique identity
- 2) has a description of its structure or the information that is used to create it
- 3) has a state representing its current condition, e.g. values of some of its features

## Behaviour:

- 1) what can the object do?
- 2) what can be done to it?

# Example: Object Definition



## 1) **information:**

- a) serial number
- b) model
- c) speed
- d) memory
- e) status

## 2) **behaviour:**

- a) print file
- b) stop printing
- c) remove file from queue

# Classes

---

What is a **class**?

- 1) any uniquely identified abstraction of a set of logically related instances that share the same or similar characteristics
- 2) rules that define objects
- 3) a definition or template that describes how to build an accurate representation of a specific type of objects
- 4) examples: agency, citizen, car, ...

Objects are instantiated (created) using class definitions as templates.



# Attributes

---

## Definition

**Attribute** is a named property of a class that describes a range of values that instances of the class may hold for that property.

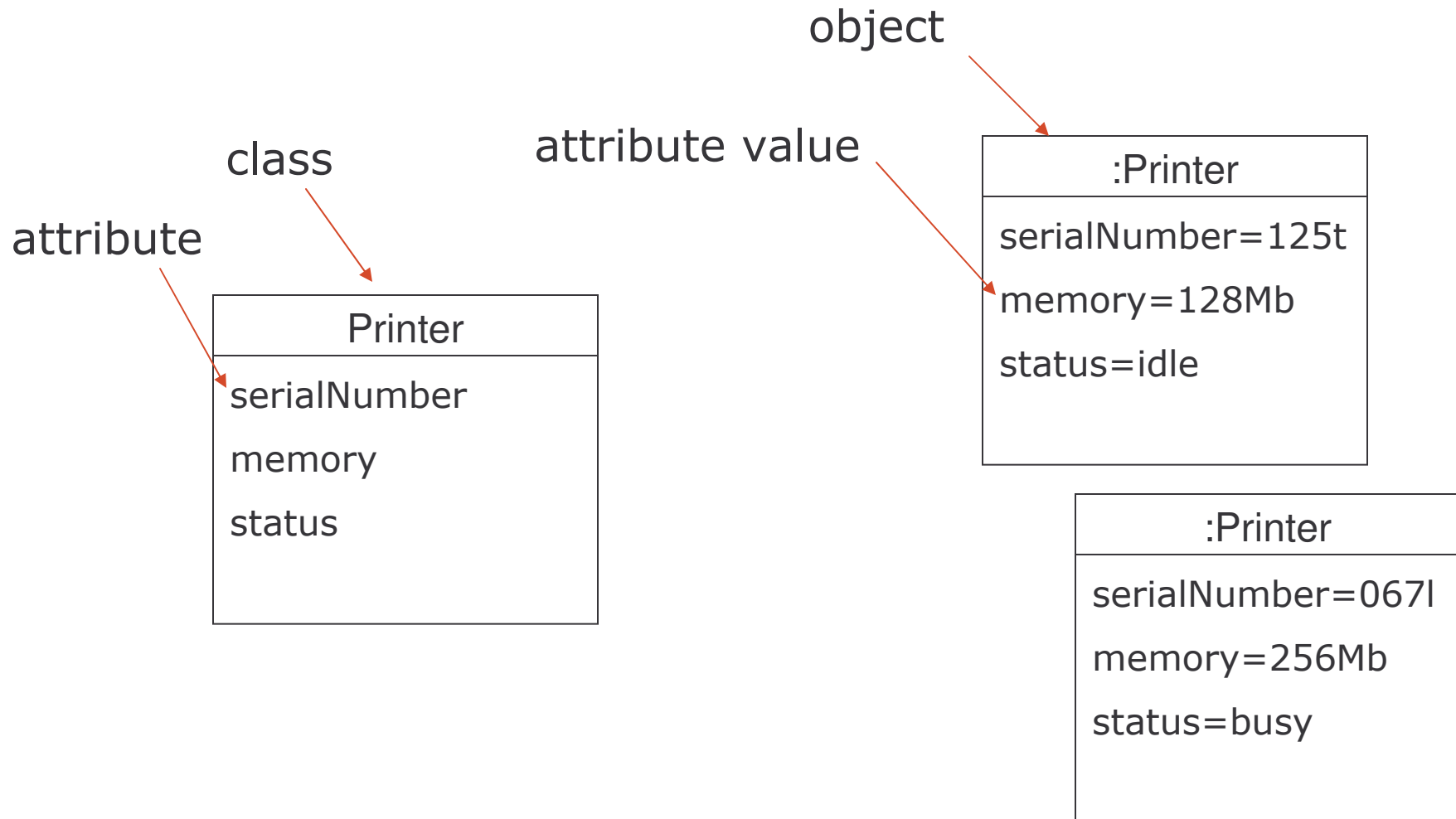
An attribute has a type and defines the type of its instances.

Only the object itself should be able to change the values of its attributes.

The given set of values of the attributes defines the state of the object.



# Example: Attributes



# Operations

---

## Definition

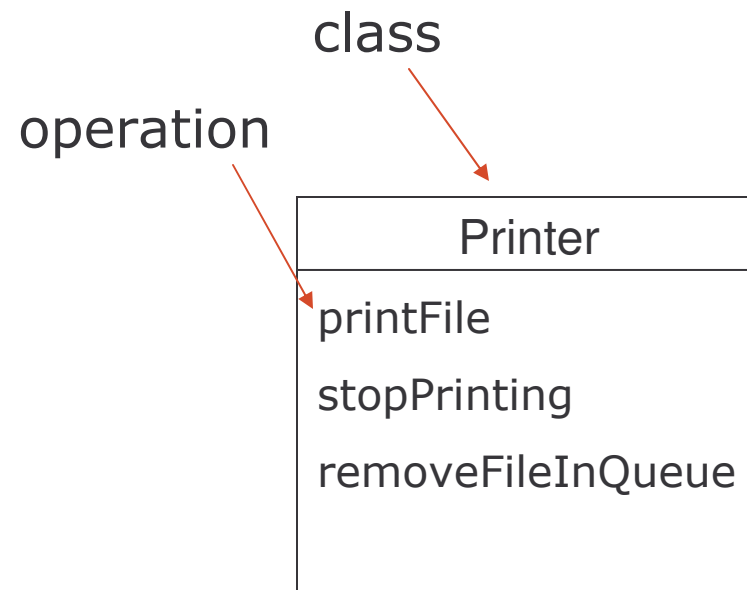
**Operation** is the implementation of a service that can be requested from any object of the class.

An operation could be:

Question - does not change the values of the object

Command - may change the values of the object

# Example: Operations



# Applying Abstraction

Abstraction in Object Orientation:

- 1) use of objects and classes in representing reality
- 2) software manages abstractions based on the changes occurring to real-world objects



*Courtesy: XML Bible*

# Encapsulation 2

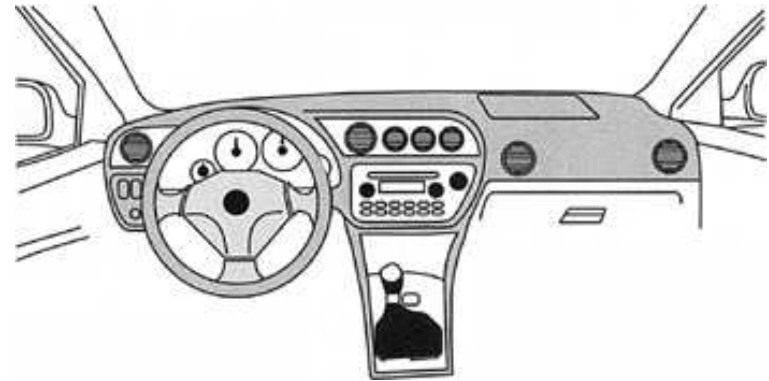
---

- 1) hallmark of object orientation
- 2) behaviour is defined once and stored inside objects
- 3) emphasizes two types of information:
  - a) interface information - minimum information for using the object
  - b) implementation information - information required to make an object work properly

# Interface

---

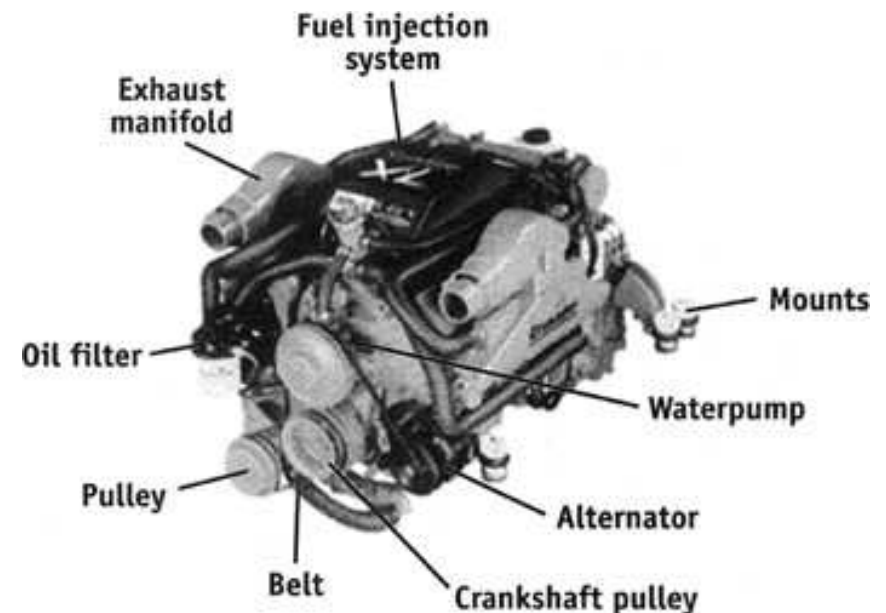
- 1) minimum information required to use an object
- 2) allows users to access the object's knowledge
- 3) must be exposed
- 4) provides no direct access to object internals



# Implementation

---

- 1) information required to make an object work properly
- 2) a combination of the behaviour and the resources required to satisfy the goal of the behaviour
- 3) ensures the integrity of the information on which the behaviour depends



# Encapsulation Requirements

- 1) to expose the purpose of an object
- 2) to expose the interfaces of an object
- 3) to hide the implementation that provides behaviour through interfaces
- 4) to hide the data within an object that defines its structure and supports its behaviour
- 5) to hide the data within an object that tracks its state



# Encapsulation Benefits

- separation of an **interface** from **implementation**, so that one interface may have multiple implementations
- data held within one object cannot be corrupted by other objects

# Associations and Links

---

## Association:

- expresses relationships between classes
- defines links between instances of classes (objects)

## Link:

- 1) expresses relationships between objects

There are different kinds of relationships between classes:

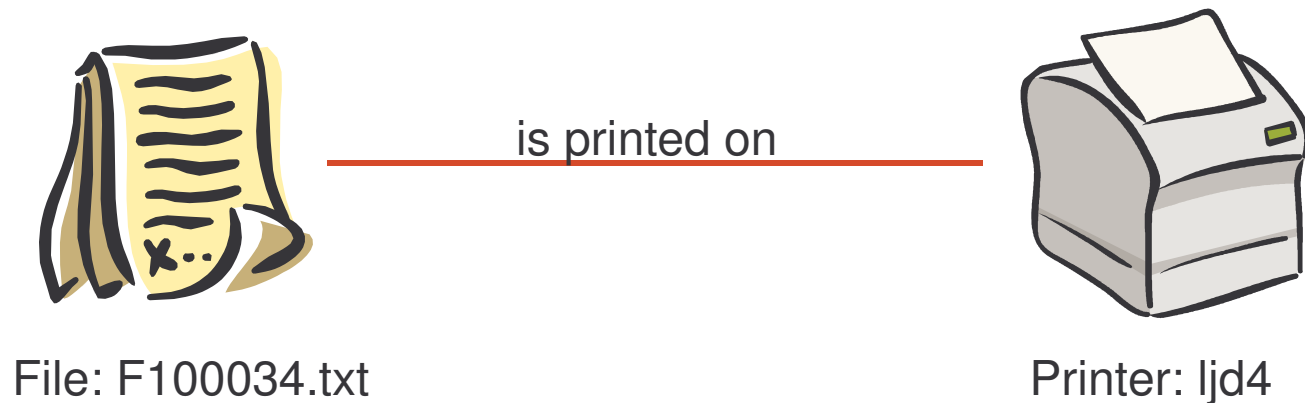
- 1) association
- 2) aggregation
- 3) composition

# Example: Associations - Links

a) Association:



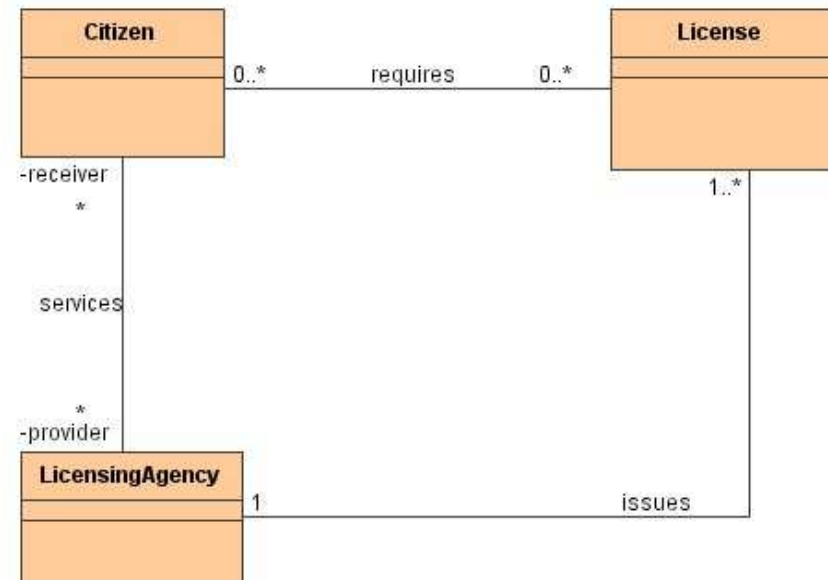
b) Link:



# Relationship: Association

Features:

- 1) the simplest form of relationship between classes
- 2) a peer-to-peer relationship
- 3) one object is generally aware of the existence of other object
- 4) implemented in objects as references



# Example: Associations

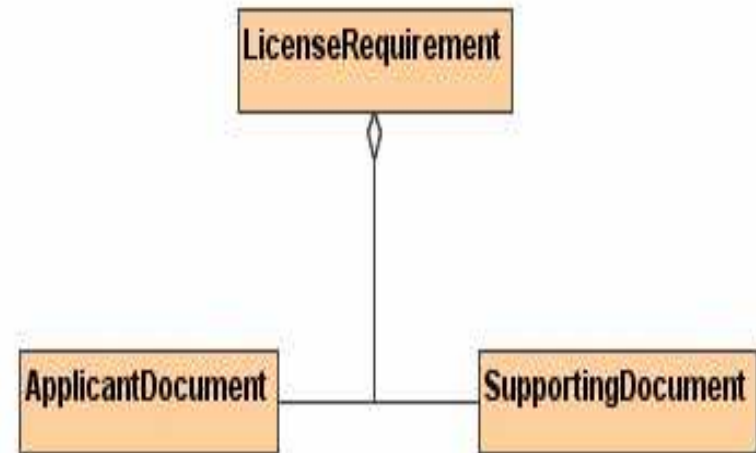
Association between classes A and B:

- 1) A is a physical/logical part of B
- 2) A is a kind of B
- 3) A is contained in B
- 4) A is a description of B
- 5) A is a member of B
- 6) A is an organization subunit of B
- 7) A uses or manages B
- 8) A communicates with B
- 9) A follows B
- 10) A is owned by B
- 11)...

# Relationship: Aggregation

Features:

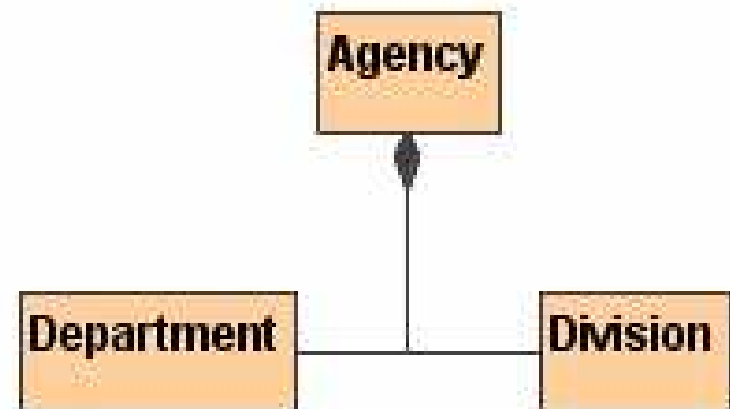
- 1) a more restrictive form of “part-of” association
- 2) objects are assembled to create a new, more complex object
- 3) assembly may be physical or logical
- 4) defines a single point of control for participating objects (parts)
- 5) the aggregate object coordinates its parts



# Relationship: Composition

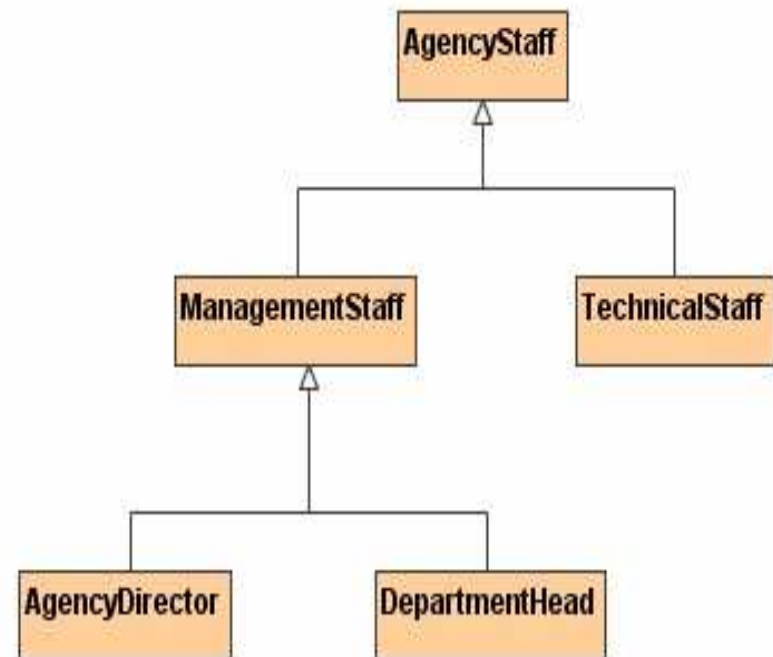
Features:

- 1) a stricter form of aggregation
- 2) lifespan of parts depend on the lifespan of the aggregate object
- 3) parts cannot exist on their own
- 4) there is a create-delete dependency of the parts on the whole



# Inheritance or Generalization

- 1) a process of organizing the features of different kinds of objects that share the same purpose
- 2) equivalent to “kind-of” or “type-of” relationship
- 3) also referred to as inheritance
- 4) specialization is the opposite of generalization
- 5) not an association!





# Super-Class and Sub-Class

## Definition

**Super-class** is a class that contains features that are common to two or more classes.

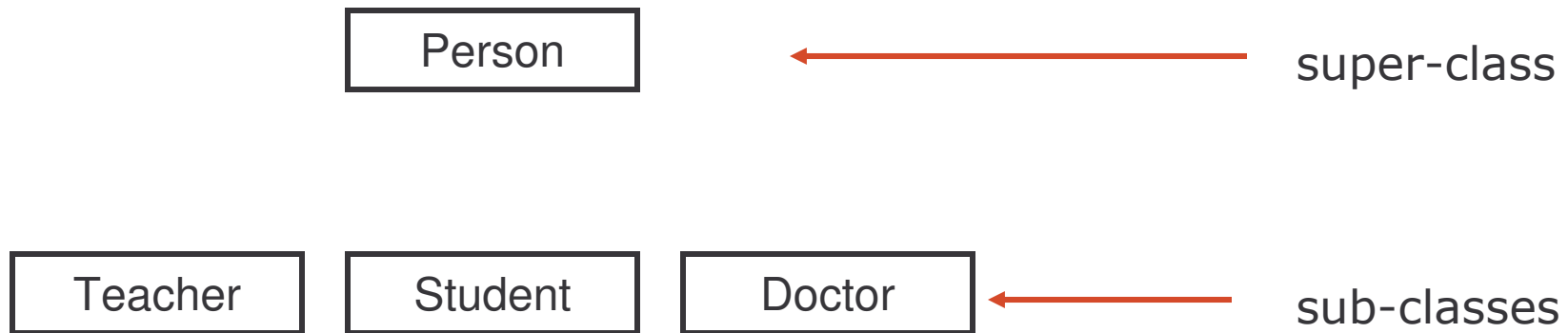
A super-class is similar to a superset. e.g. agency-staff.

## Definition

**Sub-class** is a class that contains features of its super-class or super-classes, and perhaps more.

A class may be a sub-class and a super-class at the same time. e.g. management-staff.

# Example: Super- and Sub-Class



# Abstract and Concrete Class

## Abstract class

- 1) a class that lacks a complete implementation
- 2) provides operations without implementing methods
- 3) cannot be used to create objects, i.e. cannot be instantiated
- 4) a concrete sub-class must provide methods for unimplemented operations

## Concrete class

- 1) has methods for all operations
- 2) can be instantiated
- 3) methods may be defined in the class or inherited from a super-class

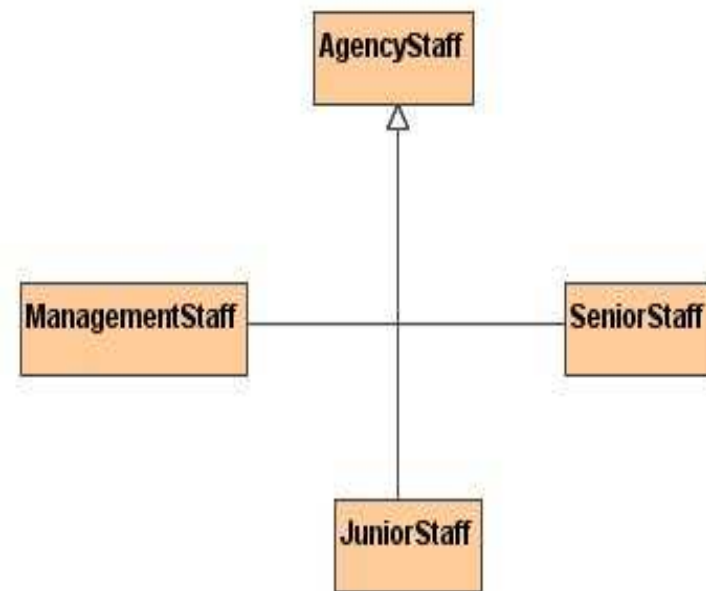
# Final / Leaf Class

---

## Leaf class

- 1) cannot have any sub-classes
- 2) a leaf node of the generalization hierarchy

Example: management staff, senior staff and junior staff are final or leaf classes.



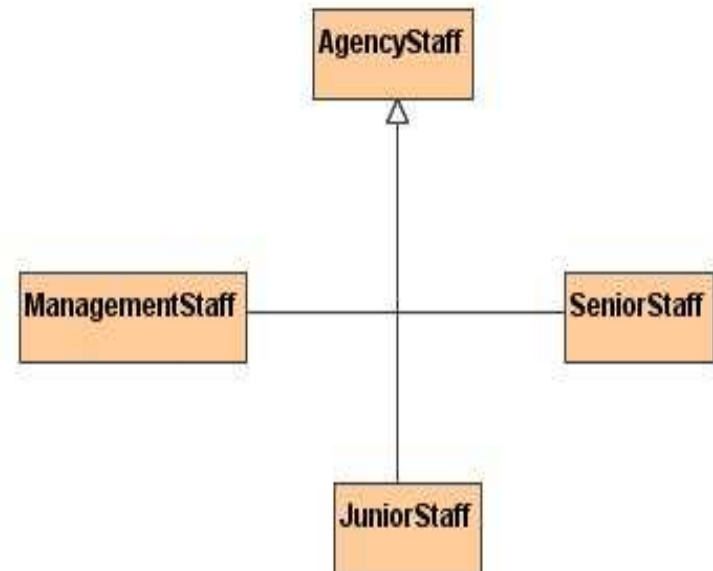
# Discriminator

---

## Discriminator

attributes that define sub-classes

Example: "status" of agency staff is a possible discriminator to derive "management", "senior" and "junior" sub-classes



# Discrimination Criteria

- 1) attribute type
- 1) attribute values allowed
- 2) operation or interface
- 3) method or implementation
- 4) association

# Polymorphism

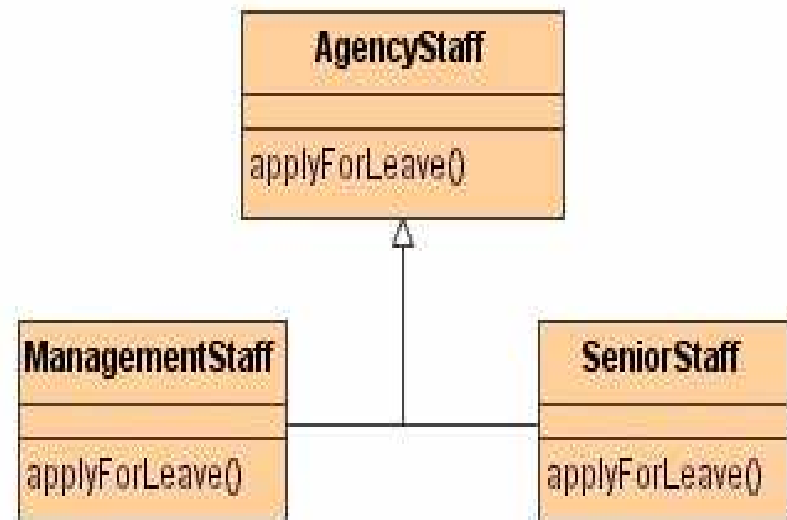
What is it?

- 1) ability to dynamically choose the method for an operation at run-time or service time
- 2) facilitated by encapsulation and generalization:
  - a) **encapsulation**: separation of interface from implementation
  - b) **generalization**: organizing information such that the shared features reside in one class and unique features in another
- 3) therefore: operations could be defined and implemented in a super-class, but the re-implemented methods are in unique sub-classes.

# Example: Polymorphism

Many ways of doing the same thing!

Example: management-staff and agency-staff can apply for leave, but possibly in different ways

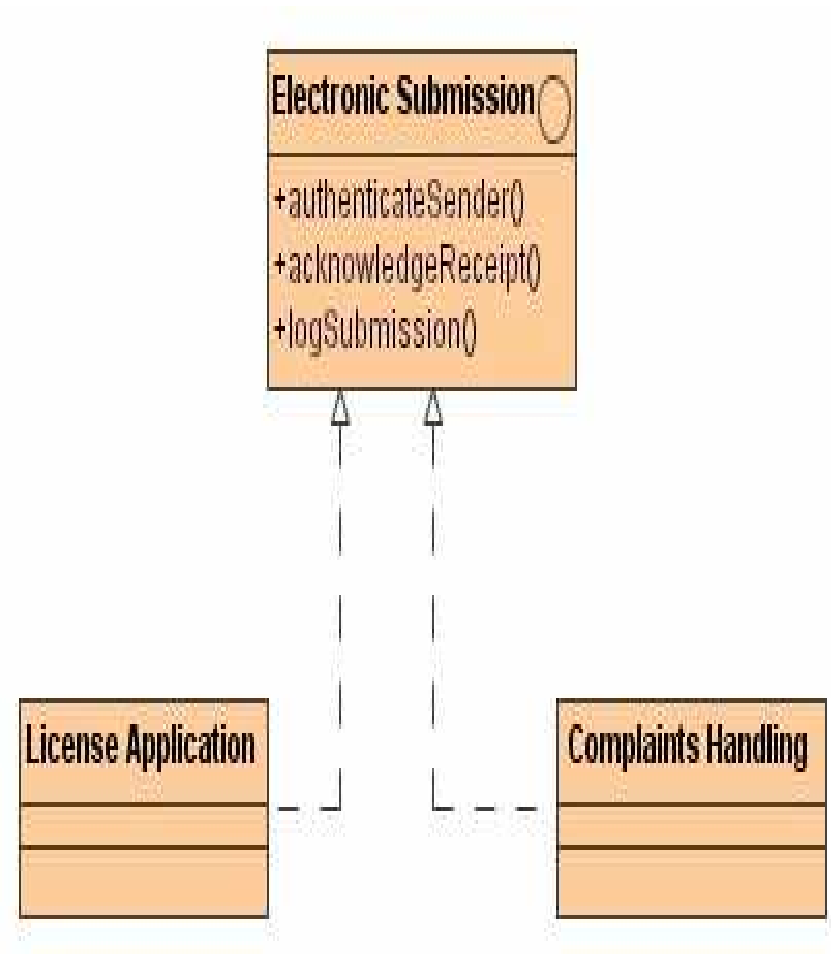




# Realization

## Realization

- 1) allows a class to inherit from an interface class without being a sub-class of the interface class
- 2) only inherits operations
- 3) cannot inherit methods, attributes, and associations



# Object Oriented Analysis

# Object Oriented Analysis 1

- 1) a discovery process
- 2) clarifies and documents the requirements of a system
- 3) focuses on understanding the problem domain
- 4) discovers and documents the key classes in the problem domain
- 5) concerned with developing an object-oriented model of the application domain
- 6) identified objects reflect the entities that are associated with the problem to be solved

# Object Oriented Analysis 2

## Definition [SEI]

Object Oriented Analysis (OOA) is concerned with developing software requirements and specifications expressed as a system's object model (composed of a population of interacting objects), as opposed to the traditional data or functional views of systems.

# Benefits

---

- 1) **maintainability**: simplified mapping to the real world
  - a) less analysis effort
  - b) less complexity in system design
  - c) easier verification by the user
- 2) **reusability**: reuse of the analysis artifacts that are independent of the analysis method or programming language
- 3) **productivity**: direct mapping to features of OOP languages

# Object Oriented Design

# Object Oriented Design 1

- process of invention and adaptation
- creates abstraction and mechanisms necessary to meet the system's behavioural requirements determined during analysis
- language independent
- provides an object-oriented model of a software system to implement the identified requirements

# Object Oriented Design 2

Definition [SEI]

Object Oriented Design (OOD) is concerned with developing an object-oriented model of a software system to implement the requirements identified during OOA.

The benefits are the same as those in OOA.



# Process

---

## Stages in OOD:

- 1) understand and define the context and modes of use of the system
- 2) design the system architecture
- 3) identify the principal objects in the system
- 4) develop design models
- 5) specify object interfaces

## Note on OOD activities:

- 1) activities are not strictly linear but interleaved
- 2) back-tracking may be done a number of times due to refinement or availability of more information

# Design Principles

---

## Cohesion and coupling

- 1) module attributes
- 2) cohesion - how tightly bound are the internal elements of different modules
- 3) coupling – to what extent one module is connected with others
- 4) for reusability, modules should have **high cohesion** and **low coupling**

# Summary

# Summary 1

---

Object is any abstraction that models a single thing in a universe with some properties and behaviour.

A class is any uniquely identified abstraction of a set of logically related objects that share similar characteristics.

Classes may be related by the following types of relationships:

- 1) association
- 2) aggregation
- 3) composition

# Summary 2

---

Object Orientation is characterized by three fundamental principles:

- 1) **encapsulation** – combination of data and behaviour, information hiding, separation of an interface from implementation
- 2) **inheritance** – generalization and specialization of classes, forms of hierarchy
- 3) **polymorphism** – different implementations for the shared operation depending on the particulars of the involving object in the inheritance hierarchy.

Object Oriented Analysis is concerned with creating requirements specifications and analysis models of the application domain.

Object Oriented Design is concerned with implementing the requirements identified during OOA, in the solution domain.

# Exercise

---

- 1) Consider any application. Describe it very briefly in text.
- 2) List five main objects for this domain.
- 3) Identify at least five classes in the application describing the types of objects mentioned in point 2.
- 4) Provide concrete examples of the association, aggregation and composition relationships in the domain.
- 5) Show how one of the identified classes can be specialized or generalized.
- 6) Explain how encapsulation and polymorphism can aid reusability.

# UML Basics

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary



# UML Basics

## Modelling Principles

# UML Basics

---

## 1) Modelling Principles

### 2) UML Overview:

- a) Goals of UML
- b) Brief history of UML
- c) Language architecture

### 3) Building Blocks:

- a) Elements: Structural, Behavioural, Grouping and Annotation
- b) Relationships

### 3) Building Blocks:

#### c) Diagrams:

- Class
- Object
- Use case
- Interaction: Sequence and Collaboration
- Statechart
- Activity
- Component
- Deployment

### 4) Views

# What Is Modelling?

- 1) representation or simplification of reality
- 2) provides a blueprint of a system
- 3) includes elements with broad effects and omits those not relevant at a given level of abstraction

# Why Modelling?

- 1) to better understand the system we are developing
- 2) to provide a model of the structure or behaviour of the system
- 3) to experiment by exploring multiple solutions
- 4) to furnish abstraction for managing complexity
- 5) to document the design decisions
- 6) to visualize the system "as-is" and "to-be"
- 7) to provide a template for constructing a system

# Why Modelling Graphically?

- 1) Graphics reveal data
- 2) 1 bitmap = 1 megaword [anonymous visual modeler]
  - Courtesy Cris Kobryn – Introduction to UML

# Modelling Principles

- 1) the choice of models has a profound influence on how a problem is attacked and how the solution is shaped
- 2) every model may be expressed at different levels of abstraction (precision)
- 3) effective models are connected to reality
- 4) No single model is sufficient. Non trivial systems are best described with a set of independent but related models

# UML Basics

## UML Overview

# UML Basics

---

## 1) Modelling Principles

## 2) UML Overview:

a) Goals of UML

b) Brief history of UML

c) Language architecture

## 3) Building Blocks:

a) Elements: Structural,  
Behavioural, Grouping  
and Annotation

b) Relationships

## 3) Building Blocks:

### c) Diagrams:

- Class
- Object
- Use case
- Interaction: Sequence  
and Collaboration
- Statechart
- Activity
- Component
- Deployment

## 4) Views



# What Is The UML?

---

- 1) UML is a language for visualizing, specifying, constructing and documenting artifacts of software intensive systems.
- 2) Examples of artifacts: requirements, architecture, design, source code, test cases, prototypes, etc.
- 3) UML is suitable for modelling various kinds of systems: enterprise information systems, distributed web-based applications, real-time embedded system, etc.

# UML – Specification Language

- 1) provides views for development and deployment
- 2) UML is process independent
- 3) recommended for use with processes that are:
  - a) use-case driven
  - b) architecture-centric
  - c) iterative
  - d) incremental

# Goals of UML

---

- 1) provide modelers with a ready to use, expressive and visual modelling language to develop and exchange meaningful models
- 2) provide extensibility and specialization mechanisms to extend core concepts
- 3) support specifications that are independent of particular programming languages and development processes
- 4) provide a formal basis for understanding the specification language
- 5) encourage/growth of the object tools market
- 6) supports higher level of development with concepts such as components frameworks or patterns

# Brief History of UML

---

- 1) started as a unification of the Booch method and the Rumbaugh method - *Unified Method v. 0.8* (1995), and in 1996 Jacobson joined them to produce UML 0.9
- 2) UML Partners worked with the Amigos to propose UML as a standard modelling language to OMG in 1996.
- 3) in 1997, the UML partners tendered their initial proposal (UML 1.0) to OMG, and 9 months later they submitted the final proposal (UML 1.1)
- 4) minor revision is UML 1.4 adopted in May 2001, and most recent revision is UML 1.5 published in March 2003.
- 5) awaiting UML 2.0 official release.

# UML Language Architecture 1

UML language architecture was provided by OMG to align UML with other OMG's technologies.

UML architecture follows the meta-modelling architecture of OMG's Meta-Object Facility (MOF).

Four layers:

- 1) meta-metamodel layer
- 2) metamodel layer
- 3) model
- 4) user objects

# UML Language Architecture 2

---

Layer	Description	Example
Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	Class, Attribute, Operations, Component
Model	An instance of a metamodel. Defines a language to define an information domain.	Agency, AgencyCode, numberUnits, staffStrength
User object (or user data)	An instance of a model. Defines a specific information model.	<CPSP, 5, 100>

# UML Basics

## Building Blocks

# UML Basics

---

## 1) Modelling Principles

## 2) UML Overview:

- a) Goals of UML
- b) Brief history of UML
- c) Language architecture

## 3) Building Blocks:

- a) Elements: Structural, Behavioural, Grouping and Annotation
- b) Relationships

## 3) Building Blocks:

### c) Diagrams:

- Class
- Object
- Use case
- Interaction: Sequence and Collaboration
- Statechart
- Activity
- Component
- Deployment

## 4) Views



# UML Building Blocks

Three basic building blocks:

- 1) **elements**: main “citizens” of the model
- 2) **relationships**: relationships that tie elements together
- 3) **diagrams**: mechanisms to group interesting collections of elements and relationships

These building blocks will be used to represent large and small complex structures.

# Elements

---

Four basic types of elements:

- 1) structural
- 2) behavioural
- 3) grouping
- 4) annotation

They will be used to specify well-formed models.

# Structural Elements

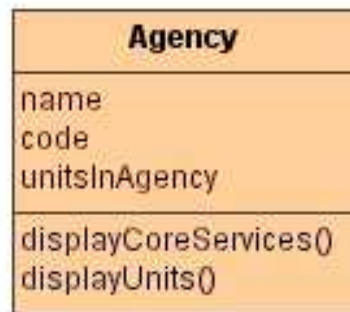
---

- 1) static part of the model to represent conceptual or physical elements
- 2) “nouns” of the model
- 3) seven kinds of structural elements:
  - a) class
  - b) interface
  - c) collaboration
  - d) active class
  - e) use case
  - f) component
  - g) node

# Class 1

---

- 1) description of a set of objects that share the same attributes, operations, relationships and semantics
- 2) implements one or more interfaces
- 3) graphically rendered as a rectangle usually including a name, attributes and operations



- 4) can be also used to represent actors, signals and utilities

# Interface

---

- 1) collection of operations that specifies a service of a class
- 1) describes the externally visible behaviour (partial or complete) of a class
- 2) defines a set of operation signatures but not their implementations
- 3) rendered as a circle with a name.



# Collaboration

---

- 1) defines an interaction between elements
- 2) several elements cooperating to deliver a behaviour rather than individual behaviour
- 3) includes structural and behavioural dimensions
- 4) represents implementations of patterns that make up a system
- 5) represented as a named ellipse drawn with a dashed line



# Use Case

---

- 1) description of a sequence of actions that a system performs to deliver an observable result to a particular actor
- 2) used to structure the behavioural elements in a model
- 3) realized by collaboration
- 4) graphically rendered as an ellipse drawn with a solid line



# Active Class

---

- 1) a class whose objects own one or more processes or threads and therefore can initiate an action
- 2) class whose objects have concurrent behaviour with other objects
- 3) graphically, an active class is rendered just like a class drawn with a thick line



- 4) it also can be used to represent processes and threads



# Component

- 1) physical replaceable part of a system that conforms to and provides the realization of a set of interfaces
- 2) represents deployment components such as COM+ or Java Beans components
- 3) represents a physical packaging of logical elements such as classes, interfaces and collaborations

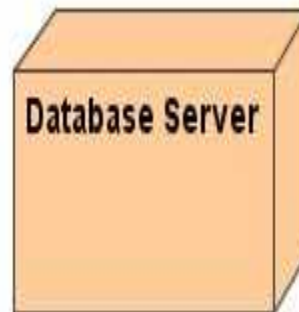


- 4) it also can be used to represent applications, files, libraries, pages and tables.

# Node

---

- 1) a physical element that exists at run time
- 2) represents a computational resource with memory and processing capacity
- 3) a set of components may reside in a node
- 4) components may also migrate from one node to another
- 5) graphically modelled as a cube.



# Behavioural Elements

- 1) represent behaviour over time and space
- 2) “verbs” of the model
- 3) two kinds of behavioural elements:
  - a) interaction
  - b) state machine

# Interaction

---

- 1) a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose
- 2) specifies the behaviour of a set of objects
- 3) involves a number of other elements:
  - messages, action sequences (behaviour invoked by a message) and links (connection between objects)
- 4) graphically rendered as an arrow

**saveapplication()**



# State Machine

---

- 1) specifies a sequence of states an object or an interaction goes through during its lifetime and its response to external events
- 2) may specify the behaviour of an individual class or a collaboration of classes
- 3) includes a number of elements including states, transition, events and activities
- 4) presented as a rounded rectangle with a name and sub-states



- 5) interactions and state machines are associated with structural elements such as classes, collaborations, and objects

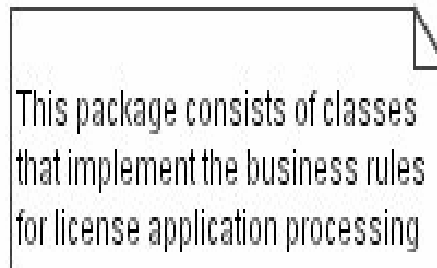
# Grouping Elements – Packages

- 1) organizational part of UML
- 2) the primary mechanism for grouping and decomposition
- 3) purely conceptual and only available at development time
- 4) graphically represented as a tabbed folder



# Annotation Elements – Notes

- 1) comments added to models for better explanation or illumination on specific elements
- 2) explanatory aspect of UML models
- 3) notes are used primarily for annotation e.g. for rendering constraints and comments attached to elements or collections of elements
- 4) presented as a rectangle with a dog-eared corner
- 5) may include both textual and graphical comments



# Relationships

---

Four basic types of relationships:

- 1) dependency
- 2) associations
- 3) generalization
- 4) realization

Meanings are consistent with the basic OO relationship types described earlier



# Relationship: Dependency

A semantic relationship between two elements in which a change to one element (independent element) may affect the meaning of the other (dependent element)

Given as a directed dashed line possibly with a label

dependent      - - - - - - - - - - ➔      independent

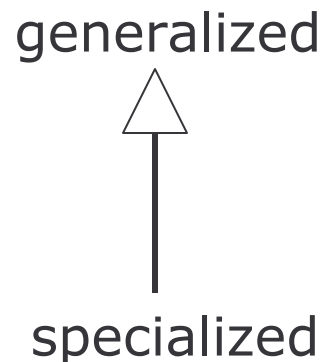
# Relationship: Association

- 1) a structural relationship describing a set of links
- 2) links are connections between objects
- 3) aggregation is a special type of association depicting whole-part relationship
- 4) association is presented as a solid line, possibly directed, labelled and with adornments (multiplicity and role names)



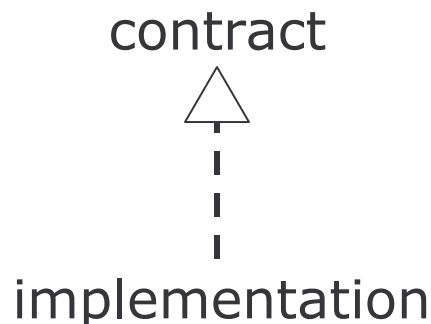
# Relationship: Generalization

- 1) a relationship in which objects of a specialized element (child) are substitutable for objects of a generalized element (parent)
- 2) child elements share the structure and behaviour of the parent
- 3) rendered graphically as a solid line with hollow arrowhead pointing to the parent



# Relationship: Realization

- 1) a semantic relationship between elements, wherein one element specifies a contract and another guarantees to carry out this contract
- 2) relevant in two basic scenarios:
  - a) interfaces versus realizing classes or components
  - b) use cases versus realizing collaborations
- 3) graphically depicted as a dashed arrow with hollow head  
a cross between dependency and generalization



# Variations to Relationships

Variations of these four relationship types include:

- 1) refinement
- 2) trace
- 3) include
- 4) extend

# Diagrams

---

- 1) a graph presentation of a set of elements and relationships where:
  - a) nodes are elements
  - b) edges are relationships
- 2) can visualize a system from various perspective thus a projection of a system
- 3) UML is characterized by nine major diagrams: a) class, b) object, c) use case, d) sequence, e) collaboration, f) statechart, g) activity, h) component and i) deployment.

# Class and Object Diagrams

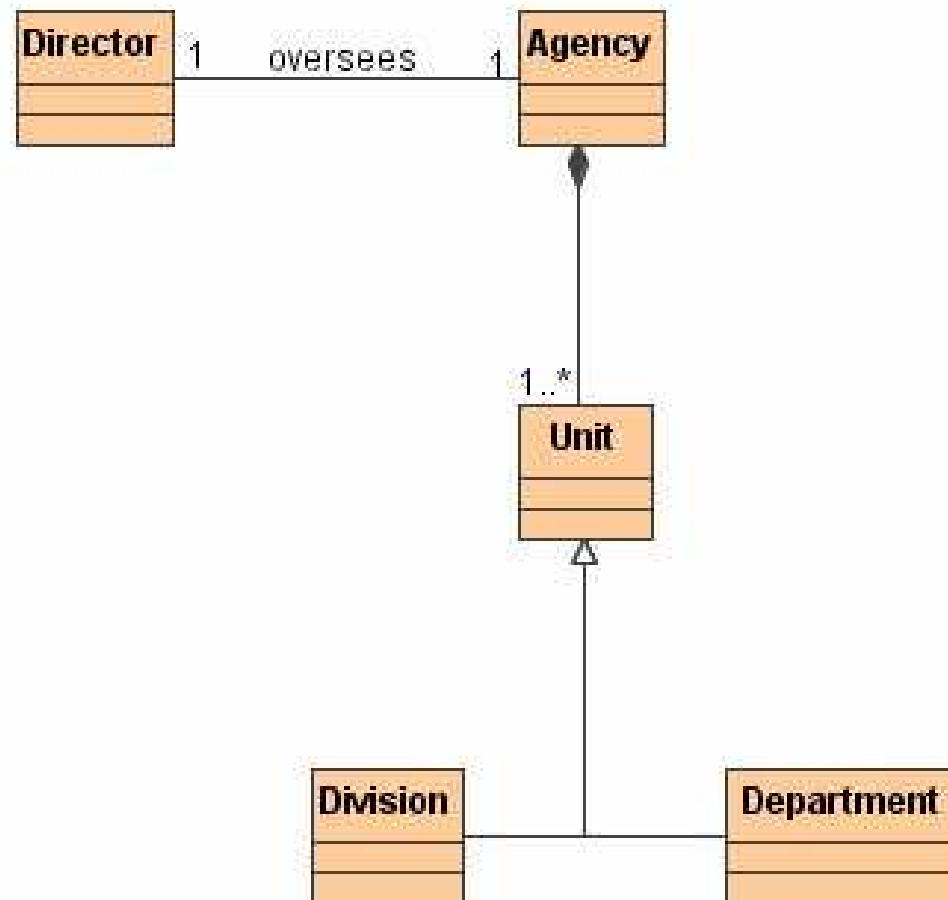
## Class Diagrams:

- 1) show a set of classes, interfaces and collaborations, and their relationships
- 2) address static design view of a system

## Object Diagrams:

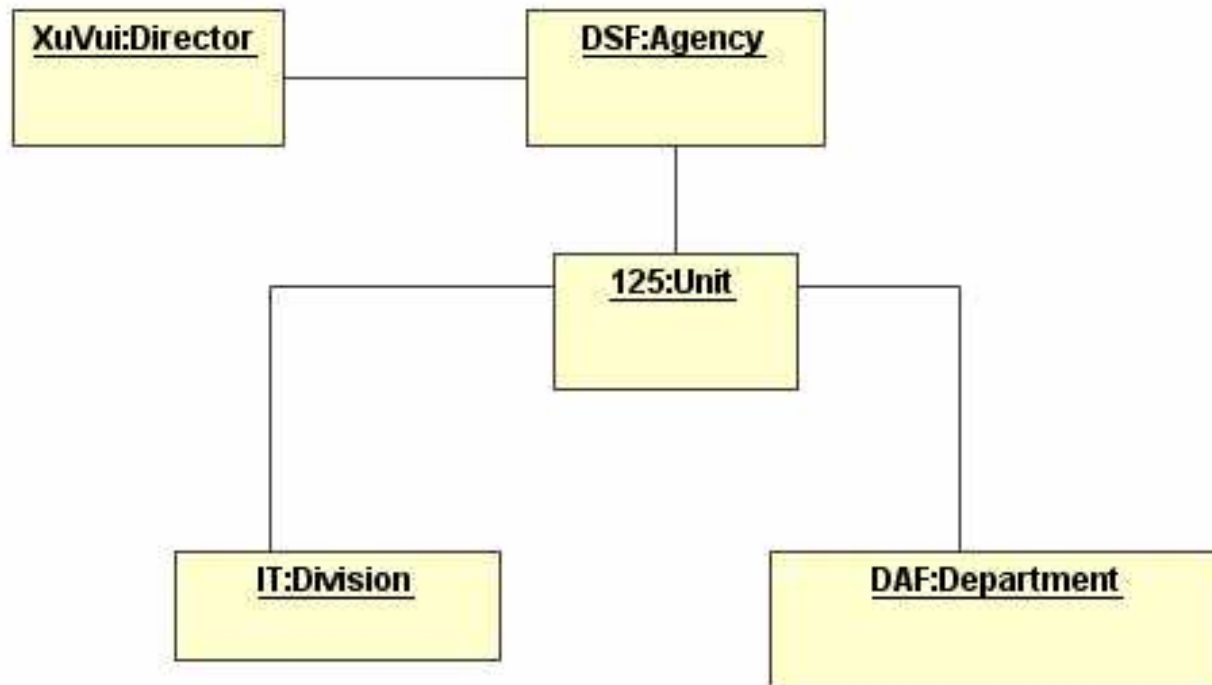
- 1) show a set of objects and their relationships
- 2) static snapshots of element instances found in class diagrams

# Example: Class Diagram





# Example: Object Diagram

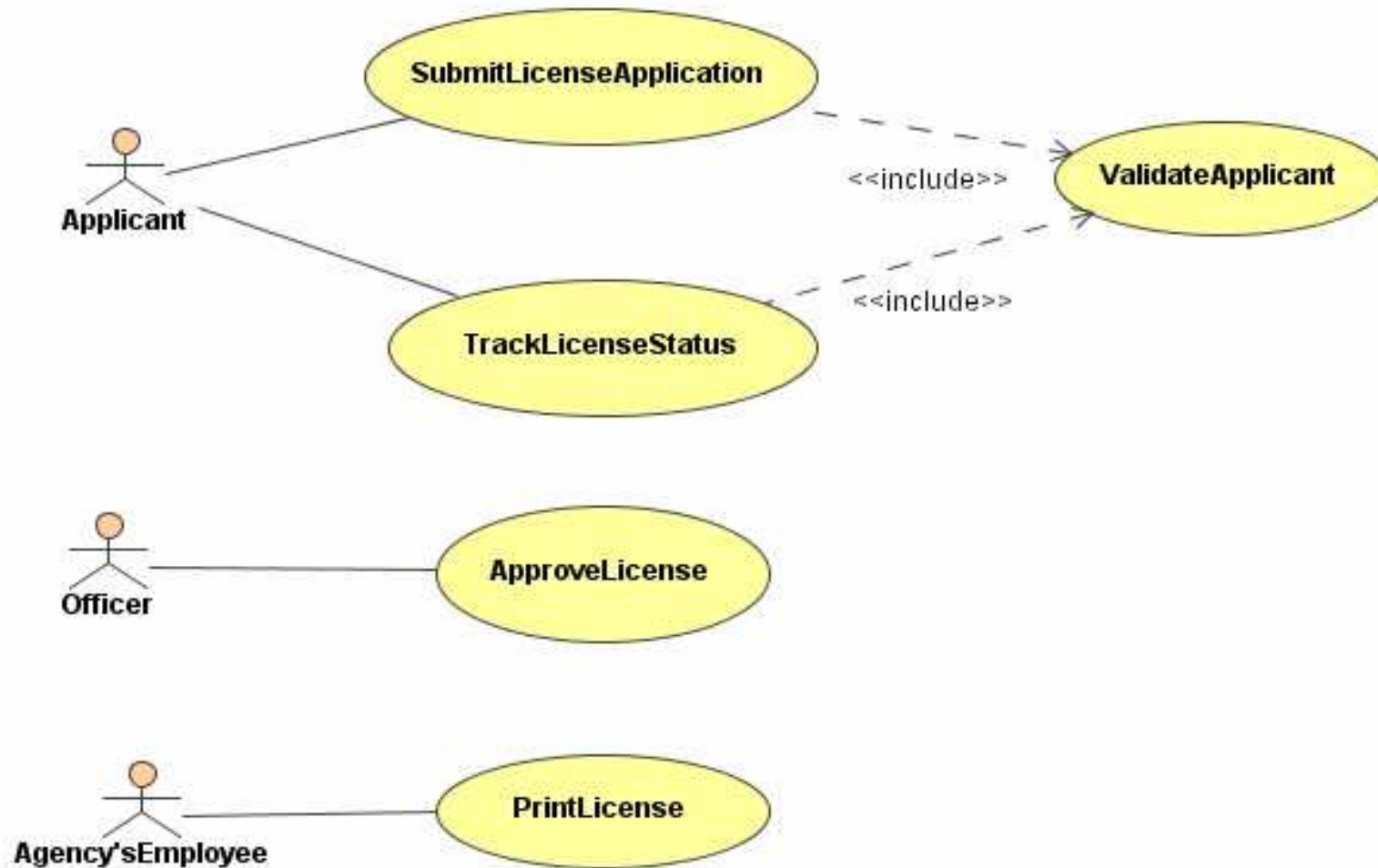


# Use Case Diagrams

---

- 1) show a set of actors and use cases, and their relationships
- 2) addresses static use case view of the system
- 3) important for organizing and modelling the external behaviour of the system

# Example: Use Case Diagram



# Interaction Diagrams

---

## Sequence Diagrams:

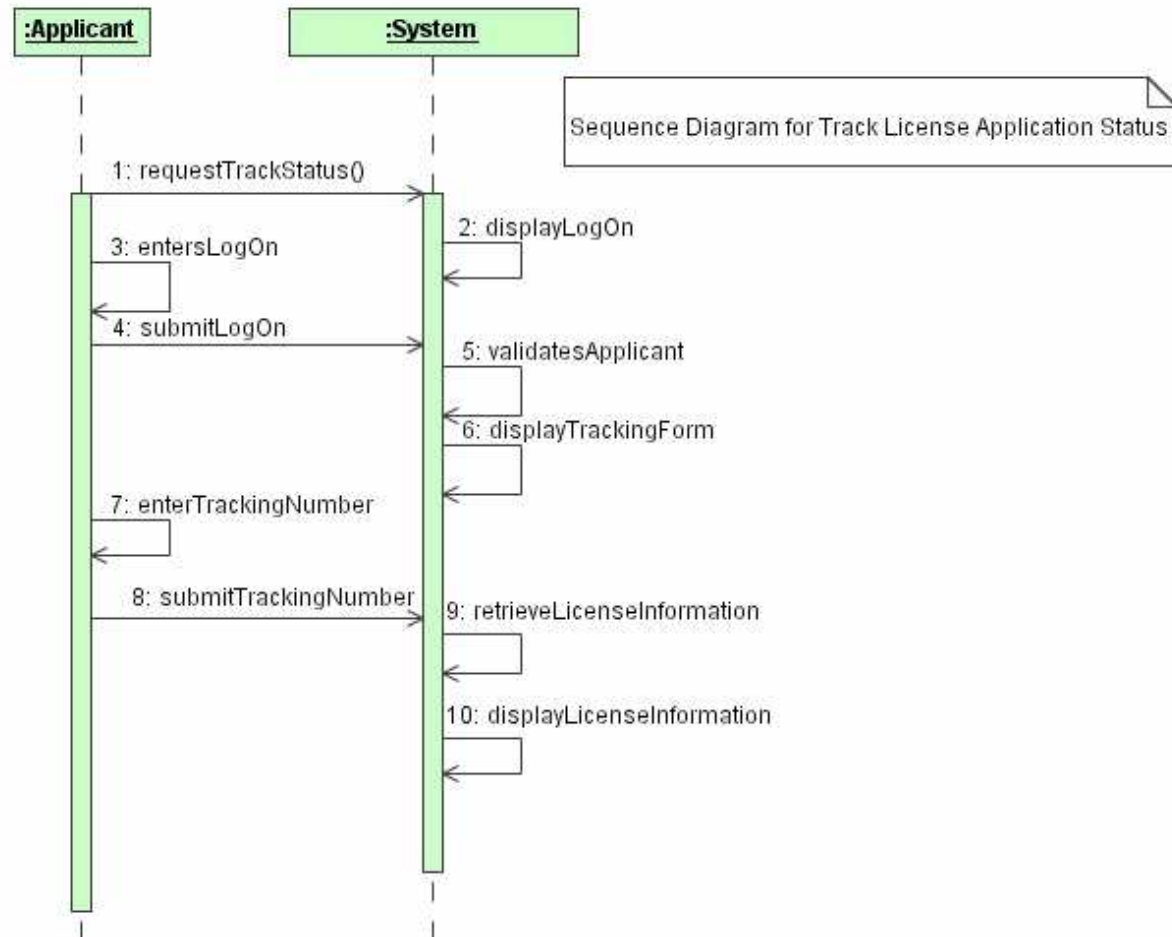
- 1) show interactions consisting of a set of objects and the messages sent and received by those objects
- 2) address the dynamic behaviour of a system with special emphasis on the chronological ordering of messages

## Collaboration Diagrams:

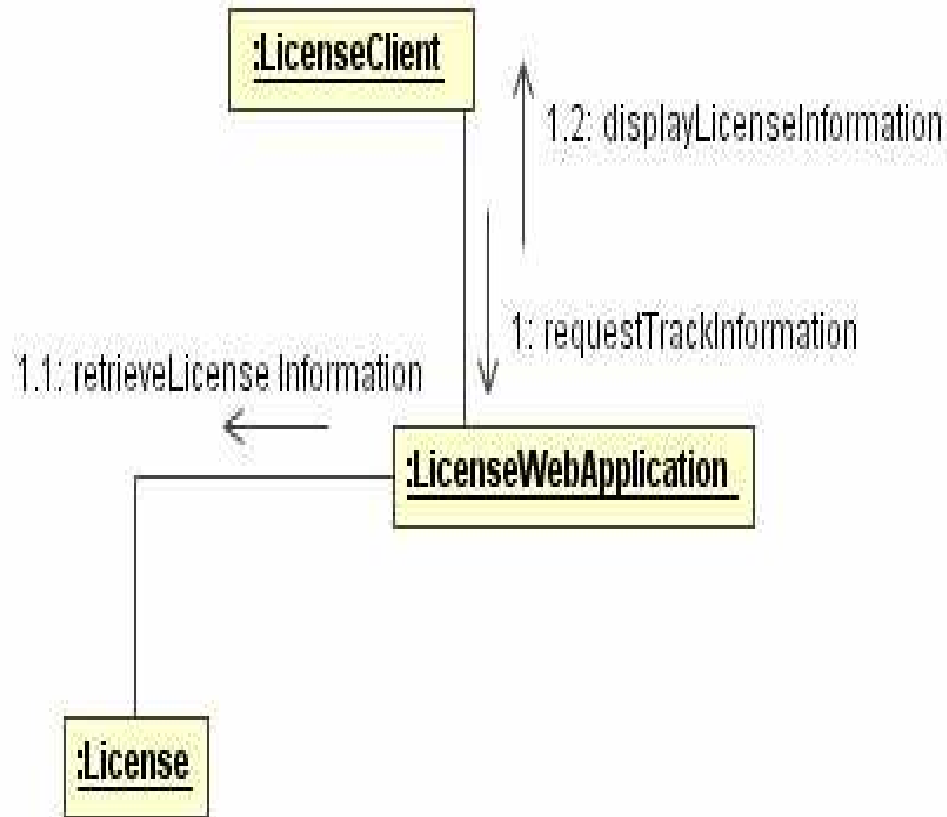
- 1) show the structural organization of objects that send and receive messages

Sequence and collaboration diagram are isomorphic i.e. one can be transformed into another

# Example: Sequence Diagram



# Example: Collaboration Diagram

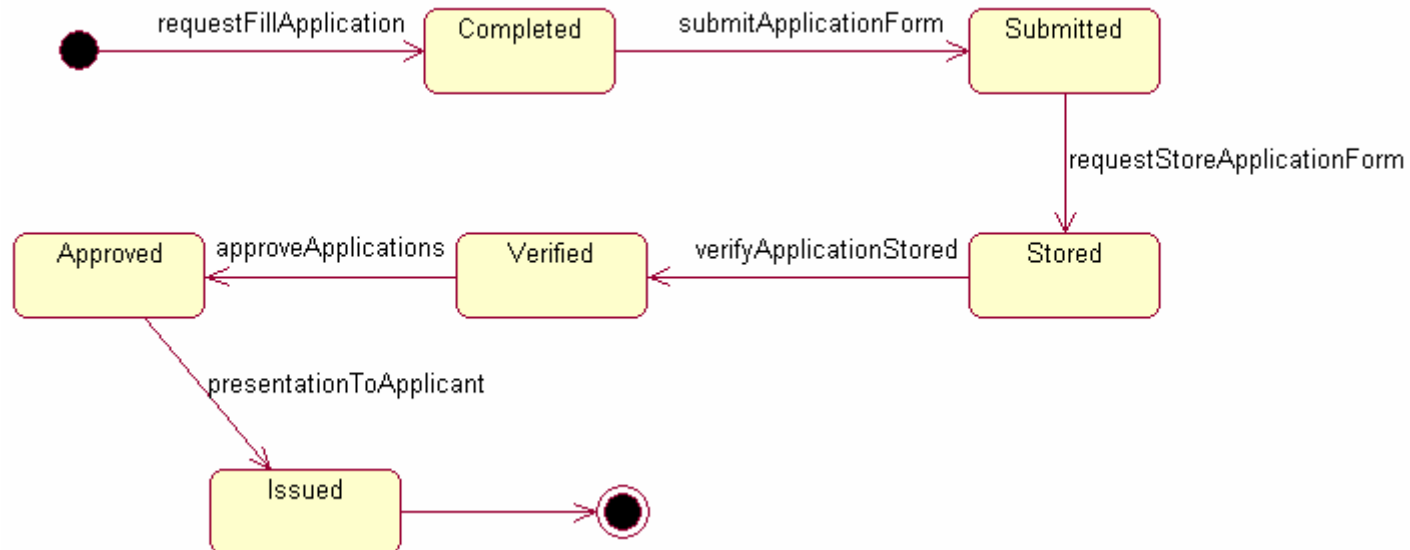


# Statechart Diagrams

---

- 1) show a state machine consisting of states, transitions, events, and activities
- 2) address the dynamic view of a system
- 3) important in modelling the behaviour of an interface, class or collaboration
- 4) emphasise on event-driven ordering

# Example: Statechart Diagram



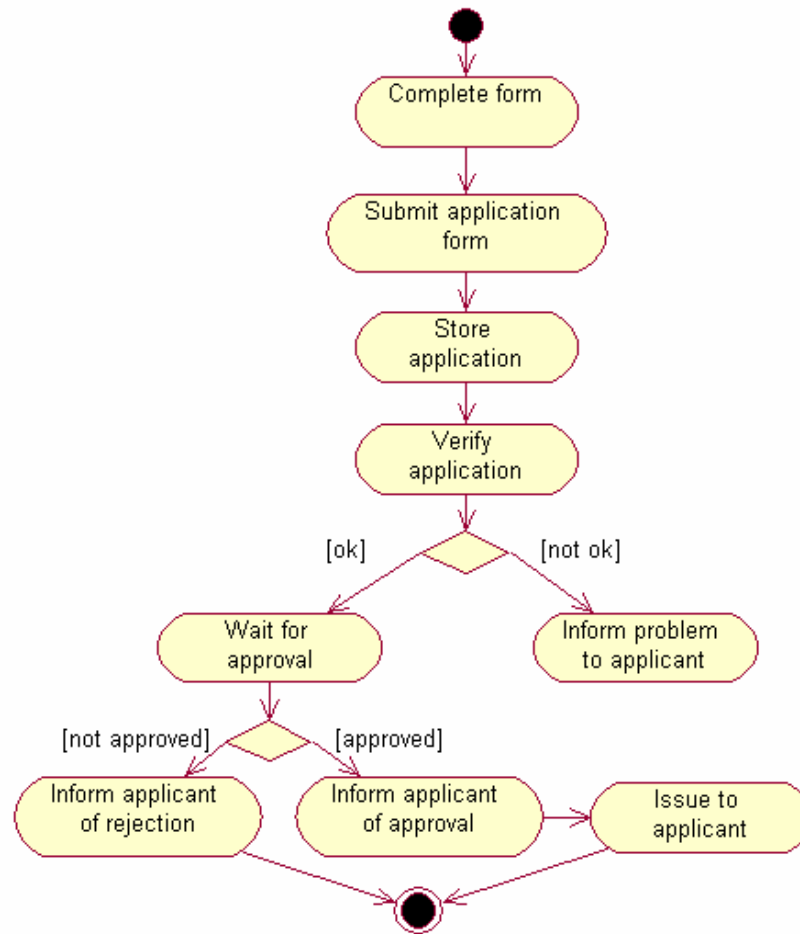


# Activity Diagrams

---

- 1) a diagram showing control/data flows from one activity to another
- 2) addresses the dynamic view of a system, useful for modelling its functions
- 3) emphasises the flow of control among objects

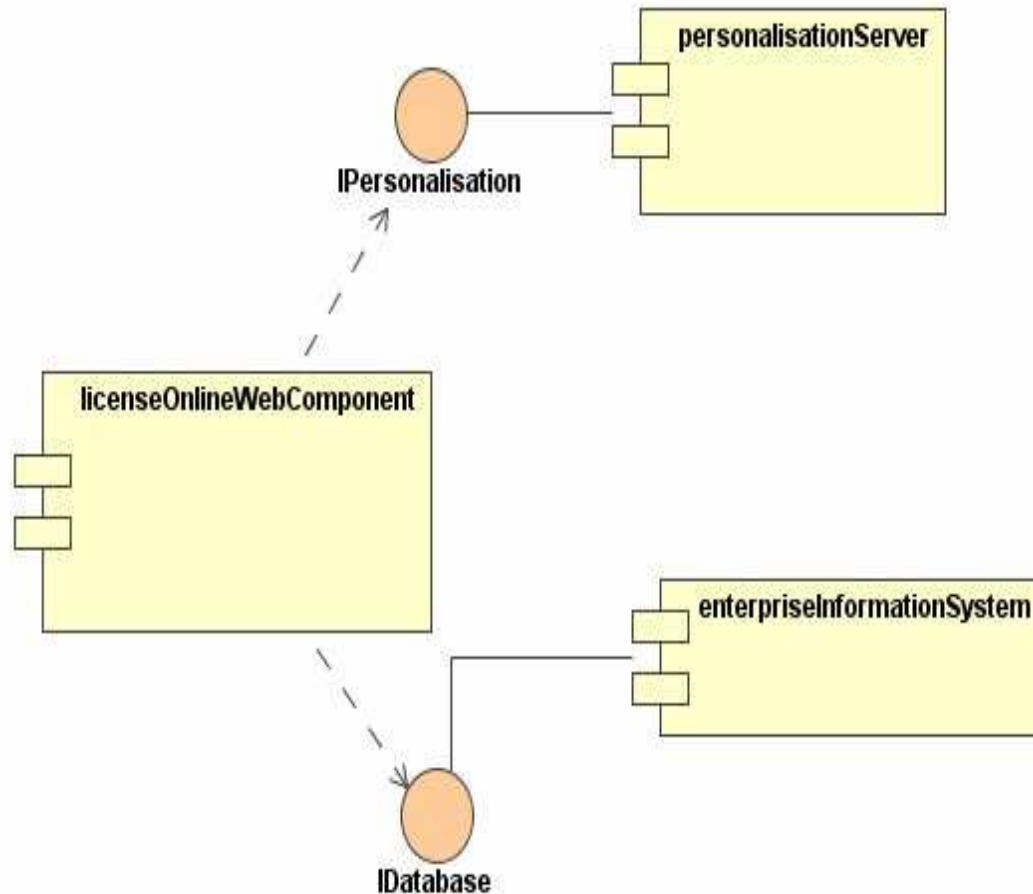
# Example: Activity Diagram



# Component Diagrams

- 1) show the organization and dependencies amongst a set of components
- 2) address static implementation view of a system
- 3) a component typically maps to one or more classes, interfaces or collaborations

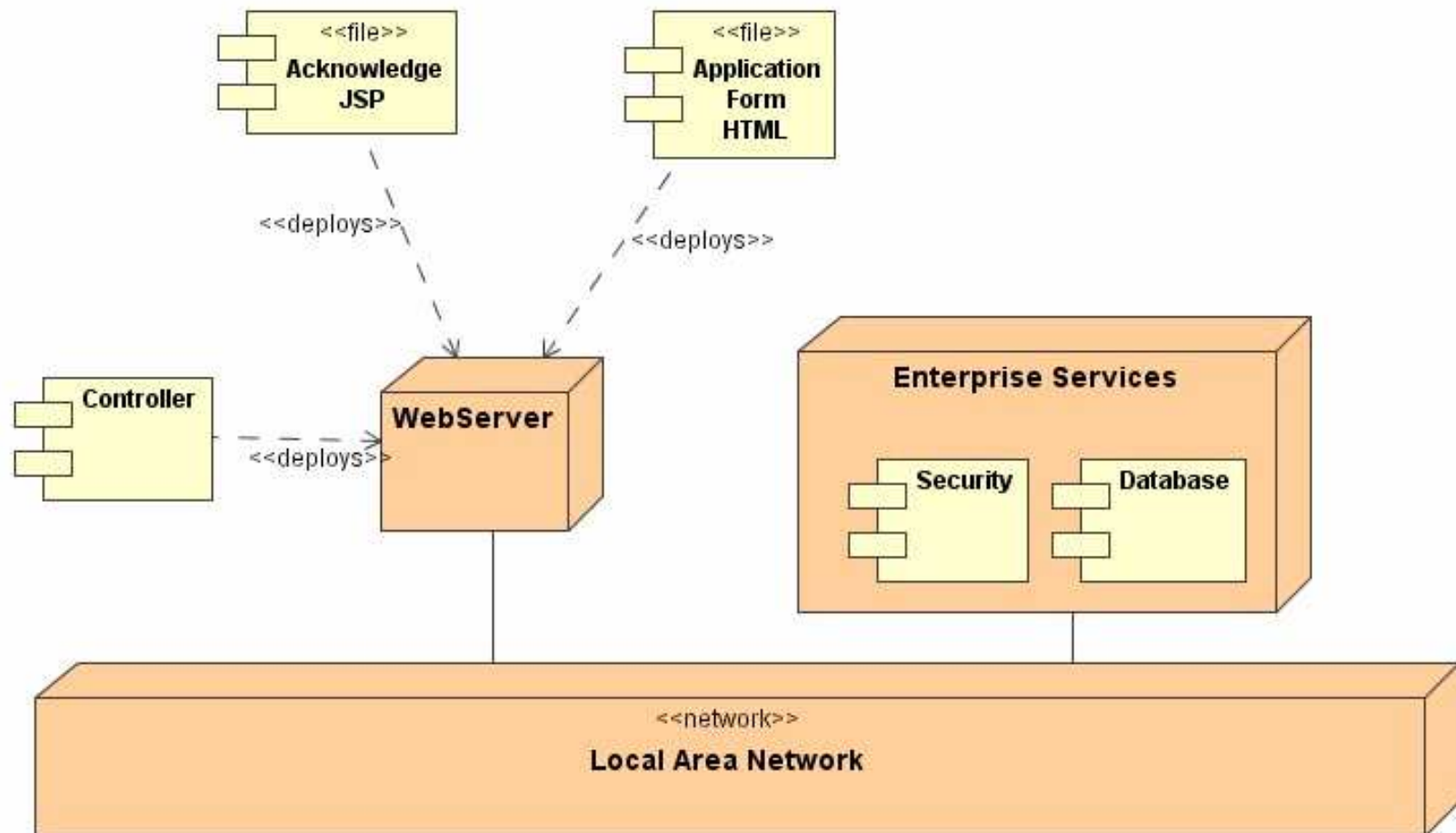
# Example: Component Diagram



# Deployment Diagrams

- 1) show configuration of run-time processing nodes and the components hosted on them
- 2) address the static deployment view of an architecture
- 3) related to component diagrams with nodes hosting one or more components

# Example: Deployment Diagram



# UML Basics

## Views

# UML Basics

---

## 1) Modelling Principles

## 2) UML Overview:

- a) Goals of UML
- b) Brief history of UML
- c) Language architecture

## 3) Building Blocks:

- a) Elements: Structural, Behavioural, Grouping and Annotation
- b) Relationships

## 3) Building Blocks:

### c) Diagrams:

- Class
- Object
- Use case
- Interaction: Sequence and Collaboration
- Statechart
- Activity
- Component
- Deployment

## 4) Views



# Modelling Views 1

---

- 1) **Use case view**: describes the behaviour of the system as seen by its end users, analysts and testers. This view shapes the system architecture.
- 2) **Design view**: encompasses the classes, interfaces, interfaces, and collaborations that form the vocabulary of the problem and its solution.
- 3) **Process view**: encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view addresses the performance, scalability and throughput of the system.

# Modelling Views 2

---

- 4) **Implementation View**: encompasses the components and files that are used to assemble and release the physical system. This view addresses the configuration management of the system's releases.
- 5) **Deployment View**: encompasses the nodes that form the system's hardware topology on which the system executes.

# Modelling Monolithic Systems

- 1) **Use case view:** use case diagrams
- 2) **Design views:** class diagrams (structural modelling) and interaction diagrams (behavioural)
- 3) **Process View:** none
- 4) **Implementation view:** none
- 5) **Deployment view:** none

# Modelling Distributed Systems

- 1) **Use case view**: use case diagrams and activity diagram (behavioural modelling)
- 2) **Design views**: class diagrams (structural modelling) interaction diagrams (behavioral modelling), statechart diagram (behavioural)
- 3) **Process View**: class diagram (structural modelling) and interaction diagrams (behavioural)
- 4) **Implementation view**: component diagrams
- 5) **Deployment view**: deployment diagrams

# Summary 1

---

A model provides a blueprint of a system.

UML is a language for visualizing, specifying, constructing and documenting artifacts of software intensive systems.

UML supports five basic views of a system: user, static-structural, dynamic, implementation and environmental modelling views.

UML is process independent but recommended for use with processes that are: use case driven, architecture-centric, iterative and incremental.

# Summary 2

---

There are three building blocks which characterize UML – elements, relationships and diagrams.

Categories of elements in UML include: structural, behavioural, grouping and annotation.

There are four basic types of relationships in UML: dependency, association, generalization and realization.

UML provides 9 diagrams for modelling: class, object, use case, sequence, collaboration, statechart, activity, component and deployment.

There are five different modelling views in UML: use case, design, process, implementation and deployment.

# Exercise

---

- 1) Why do you think visual modelling in UML is desirable?
- 2) Briefly describe the purpose of each of the UML diagrams.
- 3) Which diagrams do you consider essential in your development? Give reasons for your opinion.

# Case Study



# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# Case Study

---

This case study will be used to show examples for the different models created during the development phases.

In each model we present a particular aspect of this case study.

# e-Delivery of Licensing Services

What is a license?

- 1) a legal document or instrument that officially permits the holder to undertake some activities
- 2) required by citizens and businesses

Examples: vehicle, radio, driver, professional, building construction, business, import and export, ...

# Service Providers

---

Who provides?

- 1) government agencies, departments, offices, ...
- 2) issued by one agency or in consultation with other agencies

# Service Implementation

- 1) initiation through application, provided some basic requirements are satisfied
- 2) submission of relevant documents and information
- 3) evaluation and consultation with other agencies if necessary
- 4) decision making - issuance or denial
- 5) communication to applicant on decision

# Motivation

---

- 1) effective and efficient coordination:
  - a) several units within an agency involved
  - b) a number of other agencies may be involved
- 2) reduction of paper work: several elements move around within and outside the agency to carry out the service
- 3) citizen centered delivery: citizens may have to face several agencies just to apply for a license

# The Goal

---

Our target with this case study is:

- 1) to provide a vehicle for direct representation of the domain knowledge being acquired (UML models)
- 2) to show relevance and importance of UML to effective domain representation and software development
- 3) to demonstrate a team based approach to requirements gathering and agreement

# The Means

---

To do:

- 1) analyze the licensing service to determine some basic requirements for its electronic delivery
- 2) provide Object Oriented (specifically UML) analysis and design models for the application domain

Strategy:

- 1) concentrate on high level requirements and models to tackle the licensing services in general
- 2) models will allow room for specialization by specific agencies



# Requirements Modelling

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# Requirements Modelling: Software Requirements

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Requirements Problems

Requirements capture is a critical factor for the success of any development project.

Between 40% - 60% of all defects found in software projects can be traced back to errors made while gathering requirements.

A survey of 8000 projects undertaken by 350 US companies revealed that one-third of the projects were never completed and one-half succeeded only partially i.e. with partial functionalities, major cost over-runs and significant delays.

# Failed Projects

---

Why projects fail:

- 1) poor requirements
- 2) lack of user involvement
- 3) requirement incompleteness
- 4) changing requirements
- 5) unrealistic expectations
- 6) unclear objectives
- 7) lack of executive support

# Requirements Definition

## Definition

A requirement is

- 1) a function that a system must perform
- 2) a desired characteristic of a system
- 3) a statement about the proposed system that all stakeholders agree that must be true in order for the customer's problem to be adequately solved.

# Requirements Process

Typically includes:

- 1) **elicitation** of requirements
- 2) **modelling** and **analysis** of requirements
- 3) **specification** of requirements
- 4) **validation** of requirements
- 5) requirements **management**

The process is not linear!



# Functional and Non-Functional

## Functional requirements:

- 1) describe an interaction between the system and its environment
- 2) describe how a system should behave under certain stimuli

## Non-functional requirements:

- 1) describe the restrictions on the system that limit the choices for its construction as a solution to a given problem

# Types of Requirements

- 1) functional
- 2) interface
- 3) data
- 4) human engineering
- 5) qualification
- 6) operational
- 7) design constraints
- 8) safety
- 9) security
- 10) ...

# Requirements Specification

Requirements must be expressed formally.

Various standards are available:

- 1) IEEE P1233/D3 Guide
- 2) IEEE Std. 1233 Guide
- 3) IEEE std. 830-1998
- 4) ISO/IEC 12119-1994
- 5) IEEE std 1362-1998  
(ConOps)

Requirement specification is expected to be:

- 1) correct
- 2) consistent
- 3) feasible
- 4) verifiable
- 5) complete
- 6) traceable

# Example: Template

Requirement #:	Requirement Type:	Event/use case #:
Description:		
Rationale:		
Source:		
Fit Criteria:		
Customer Satisfaction:	Customer Disatisfaction:	
Dependencies:	Conflicts:	
Supporting Materials:		
History:		

**Volere**  
Copyright © Atlantic Systems Guild

# Example: Functional

---

Ref. Id	Description	
F1	Allow online application for license	
	F1.1.	Allow applicant download application form
	F1.2	Applicant will be able to complete license application form online
	F1.3	Applicant will be able to upload completed application form
	F1.4	System will send an acknowledgement of the receipt of application to the applicant
	F1.5	Applicant will be able to upload documents necessary for application
F2	Applicant will be able to track progress of its application	
	F2.1	Applicant will be able to determine the status of its application online
	F2.2	Applicant will be able to make enquiries on the status of its application online
	F2.3	System will inform applicant of changes in status of its application automatically

# Example: Non-Functional

Ref. Id	Description	
NF1	Applicant will be able to use the system interface with no intuition	
	NF1.1	System will provide tips on all controls (buttons etc.) available on forms
	NF1.2	Applicant will have access to context sensitive help on interface elements
	NF1.3	Explicit information on type of data and range of values acceptable for data entry fields will be available for all data entry boxes.
NF2	Applicant will be able to complete data entry forms over multiple sessions	
	NF2.1	Applicant will be able to log in and initiate sessions
	NF2.2	Applicant will be able to save a session and resume later
F3	The average burden time for the applicant in completing license application online form will be 15 minutes	

# Requirements Modelling: Use Case Modelling

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary



# Use Cases

---

- 1) describe or capture **functional requirements**
- 2) represent the desired behaviour of the system
- 3) identify users ("actors") of the system and the associated processes
- 4) are the basic building blocks of use case diagrams and use case models
- 5) tie requirements phase to other development phases

# Definition

---

A use case:

- 1) is a collection of task-related activities describing a discrete chunk of a system
- 2) is a description of a set of **actions sequences** that a system performs to obtain an **observable result** to an actor
- 3) describes the system from an external usage viewpoint

Key attributes: description, action sequence, includes variants, produces observable results

Use cases do not describe:

- 1) user interfaces
- 2) performance goals
- 3) non-functional requirements

# Example: Use Cases

**SubmitLicenseApplication**

**PrintLicense**

**ApproveLicense**

**ValidateLicense**

# Use Case Relationships 1

Use cases are organized by relationships:

1) **generalization**

- a) the same meaning as before
- b) more specialized use cases are related to more general use cases

2) **include**

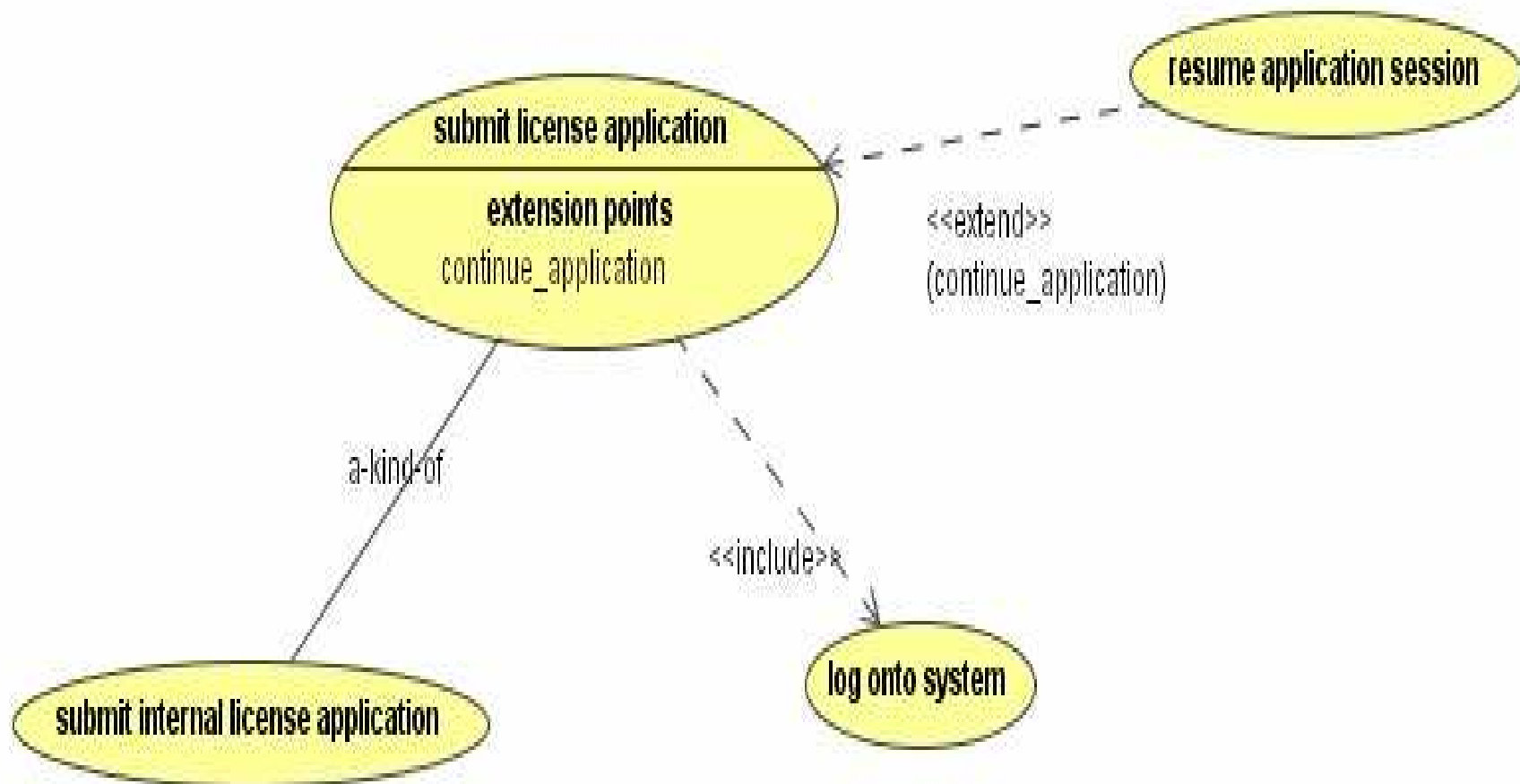
- a) the base use case explicitly incorporates the behaviour of another use case at a location specified in the base
- b) the included relationship never stands alone, but is only instantiated as part of some large base of the use cases that include it
- c) rendered as the “**include**” stereotype

# Use Case Relationships 2

## 3) **extend**

- a) the base use case implicitly incorporates the behaviour of another use case at a location specified by the extending use case (**extension point**)
- b) base use case may stand alone and usually executes without regards to extension points
- c) depending on system behaviour, the extension use case will be executed or not
- d) rendered as the “**extend**” stereotype

# Example: Use Case Modelling



# Actors

---

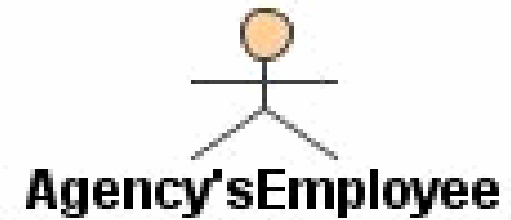
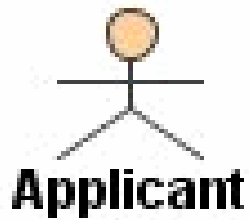
## Definition

**Actor** is anyone or anything that interacts with the system causing it to respond to business events.

Actor:

- 1) is something or somebody that stimulates the system to react or respond to its request
- 2) is something we do not have control over
- 3) represents a coherent set of roles that the entities external to the system can play
- 4) represents any type of a system's user

# Example: Actors





# Notes about Actors

---

- 1) actors **stimulate** the system with input events or **receive** something from the system
- 2) actors **communicate** with the system by sending messages to it and receiving messages from the system as they perform the use cases
- 3) actors model anything that needs to interact with the system to exchange information – human users, computer systems, etc.
- 4) a physical user may act as one or several actors as it interacts with the system, several individual users may act as different instances of one and the same actor

# Types of Actors

---

Initiator versus participant:

- When there is more than one actor in a use case, the one that generates the stimulus is called the **initiator** and the others are **participants**

Primary versus secondary:

- The actor that directly interacts with the system is called the **primary** actor, others are called **secondary** actors.

# Glossary 1

---

- 1) a set of terms that are defined and understood to form the basis for communication
- 2) is a dictionary for modelling
- 3) its purpose is to clarify the meaning of terms or to have the shared understanding of the terms amongst team members
- 4) is created during requirements definition, use case identification, conceptual modelling
- 5) is maintained throughout development

# Glossary 2

---

It is usually the central place for:

- 1) definitions of key concepts
- 2) clarification of ambiguous terms and concepts
- 3) explanations of jargons
- 4) description of business events
- 5) description of software actions

There is no specific format for glossaries:

- 1) reference identification for terms
- 2) definitions
- 3) categories
- 4) cross references

# Example: Glossary

---

Ref. Id	Name	Description	Category	Cross reference
U001	Submit License Application	Describes a functional requirement which specifies that the system must allow user submit application for license.	Use Case	CB001
A001	Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. The applicant may be an internal or an external applicant.	Actor	A002, A003
A002	External Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. This entity excludes the employees of the licensing.	Actor	A001
A003	Internal Applicant	A staff member of the agency who requires the licensing service.	Actor	A001
C1001	Agency	An administrative unit of government that provides a set of differentiated services to citizens businesses and other government agencies.	Concept	
CB001	Capture Online Submission	A structural and behavioural entity which implements the "submit application license" use case.	Collaboration	U001

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Use Case Diagrams

---

- 1) is a diagram that shows a set of use cases and actors, and their relationships
- 2) is central to modelling the behaviour of the system
- 3) is used to visualize the behaviour of a system, so that users can comprehend how to use that system, and developers can understand how to implement it
- 4) puts everything together
- 5) commonly contains:
  - a) use cases
  - b) actors
  - c) dependency, generalization and association relationships

# Identifying Use Cases

Use cases describe:

- 1) the functions that the user will want from the system to accomplish
- 2) the operations that create, read, update, delete information
- 3) how actors are notified of the changes to the internal state of the system and how they notify the system about external events



# Identifying Actors

---

To determine who are the actors, we try to answer the following questions:

- 1) who uses the system?
- 2) who gets information from the system?
- 3) who provides information to the system?
- 4) who installs, starts up or maintains the system?

# Naming Use Cases

Use **concrete verb-noun phrases**:

- 1) a weak verb may indicate uncertainty, a strong verb may clearly identify the action taken
  - a) **strong verbs**: create, merge, calculate, migrate, activate
  - b) **weak verbs**: make, report, use, copy, organize, record...
  
- 2) a weak noun may refer to several objects, a strong noun clearly identifies only one object
  - a) **strong nouns**: property, payment, transcript ...
  - b) **weak nouns**: data, paper, report, system

# Naming Actors

- 1) group individuals according to how they use the system;  
identify the roles they adopt when using the system
- 2) each role is a potential actor
- 3) name each role and define its distinguishing characteristics
- 4) not equate job titles with roles; roles cut across job titles
- 5) use common names for an existing system; avoid inventing new names

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Use Case Definition Template

Fields	Description
Use Case Name	Name of the use case
Actors	Role names of people or external entities initiating and participating in the use case
Purpose	The intention of the use case
Overview	A brief description of the usage of the process
Precondition	A condition that must hold before a use case can begin
Variation	Different ways to accomplish use case actions
Exceptions	What might go wrong during the execution of the use case
Policies	Specific rules that must be enforced by the use case
Post-conditions	Condition that must prevail after executing the use case
Priority	How important is the use case?
Frequency	How often is the use case performed?
Cross reference	Relate use cases and functional requirements

# Case Study: Use Cases

---

Use Case	Overview
Submit License Application	<p>A citizen, business or employee of a licensing agency visits the website of the licensing agency providing the online licensing service. The applicant logs-on into the site. The applicant then completes the on-line application form. This may be done over a number of sessions. Finally the applicant submits the data entered through the form into the agency license database. The applicant receives an acknowledgement of the receipt of application with an application number for tracking.</p> <p>The applicant may choose to download the application form and complete it off-line. The applicant later uploads the completed form.</p>
Track license application status	<p>The applicant who has already successfully submitted a license application logs onto the agency site. On successful entry the applicant enters the license application assigned to it. The status of its application is then displayed.</p>
Review license application documents	<p>The processing officer retrieves all relevant records associated with a particular application. The officer notifies all entities within the agency whose inputs are needed to act on the case. The officer also sends the necessary documents to all supporting agencies.</p>

# Case Study – Possible Actors

Actor	Description
External Applicant	An entity (citizen, business, visitor, agency etc.) who is applying for a license. This entity excludes the employees of the licensing agency.
Internal Applicant	A staff member of the agency who requires the licensing service
Processing Personnel	Any agency staff responsible for processing the application
Approving Authority	The entity (e.g. the agency director, sub-director or board) within the agency responsible for the final approval of the license application.
Supporting Agencies	Other agencies that provide technical inputs or opinions to the licensing agency on the license application
Licensing Agency	The agency that receives the license application and issues the license
Website visitor	Any entity (citizens, business, system, etc.) visiting the licensing website
Subscriber System	External systems that request licensing information from the license database in the licensing agency
System Administrator	IT personnel in the licensing agency responsible for administering the licensing system

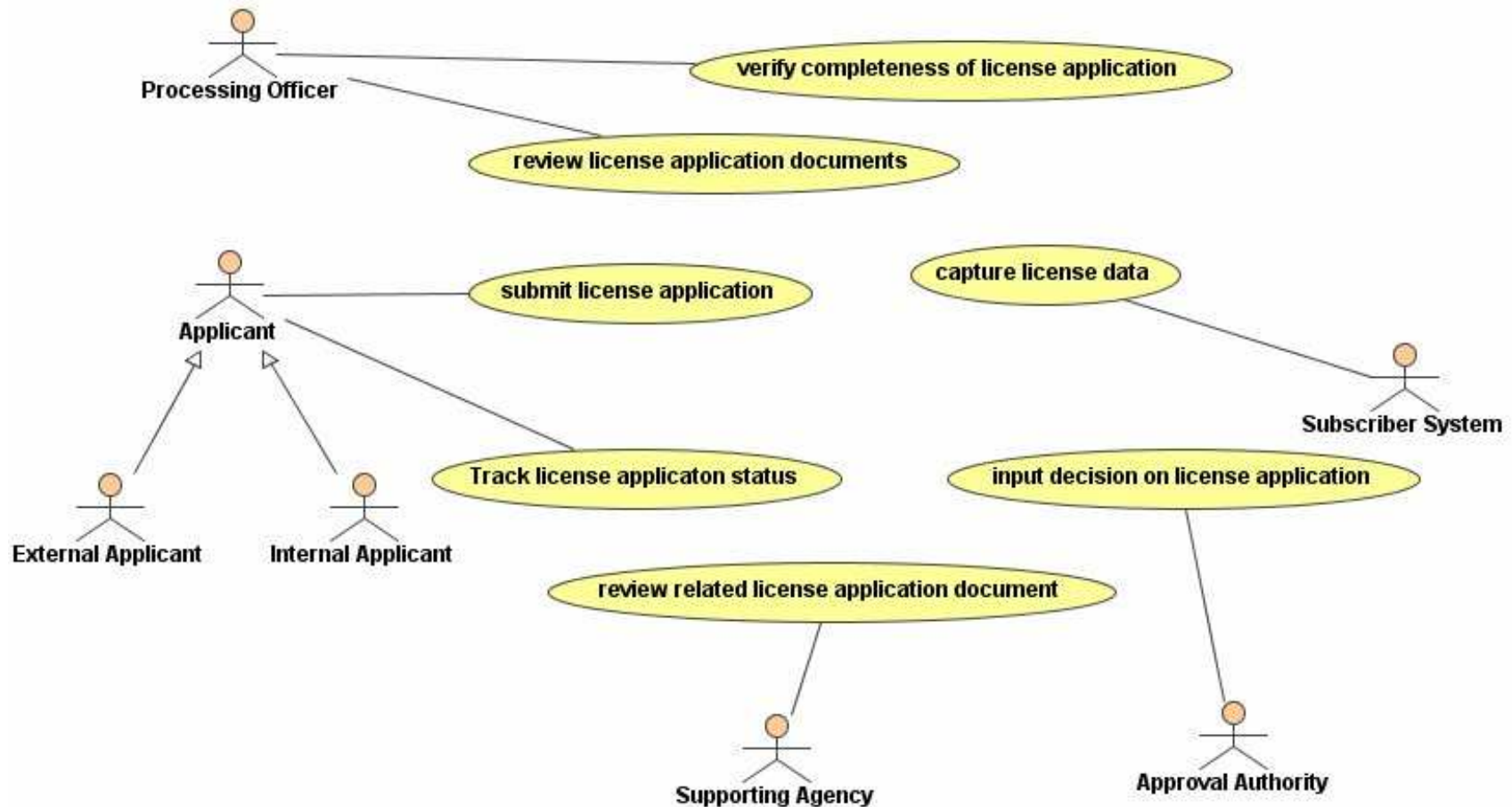
# Case Study: Description

---

Fields	Description
<b>Use Case Name</b>	Submit License Application
<b>Actors</b>	Applicant (External Applicant or Internal Applicant)
<b>Purpose</b>	Capture the online license application
<b>Overview</b>	A citizen, business or employee of a licensing agency visits the website of the licensing agency providing the online licensing service. The applicant logs-onto the site. The applicant then completes the online application form. This maybe done over a number of sessions. Finally the applicant submits the data entered through the form into the agency license database. The applicant receives an acknowledgement of the receipt of application with an application number for tracking.
<b>Precondition</b>	The Applicant fulfils the basic requirements for applying for a license
<b>Variation</b>	External applicant may submit its application at the agency location or other designated venues.
<b>Exceptions</b>	Applicant submits incomplete information
<b>Policies</b>	Acknowledgement of the receipt of application must be sent to the applicant within the time limit specified in the performance pledge relating to this service
<b>Post-conditions</b>	An acknowledgement must be sent to the applicant upon successful submission of its application and the application must be available in the license database.
<b>Priority</b>	5 (the highest). In scale 1 to 5.
<b>Frequency</b>	50 per/day
<b>Cross reference</b>	F1 (F1.1 – F1.5), NF2 (NF2.1 and NF2.2), NF3



# Case Study: Use Case Diagram



# Requirements Modelling: Conceptual Modelling

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Concept Definition

---

What is a concept ?

- an **idea**, **thing** or **object**

A concept can be:

- 1) represented symbolically
- 2) defined or described
- 3) exemplified

What is an instance?

- each concrete application of the concept

# Example: Concepts

- 1) agency
- 2) license
- 3) officer
- 4) applicant
- 5) internal applicant
- 6) external applicant

# Concept Identification

Concepts can be:

- 1) physical or tangible objects
- 2) places
- 3) documents, specifications, design or descriptions
- 4) roles of people
- 5) container of other things
- 6) organizations
- 7) processes
- 8) catalogs
- 9) ...

Concepts may be identified from requirements definitions and use cases.

# Conceptual Model and Classes

## Conceptual model:

- 1) captures the concepts in a domain in an abstract way
- 2) important part of OO requirements analysis

## Classes:

- 1) equivalent to concepts in UML
- 2) an abstraction of a set of objects
- 3) objects are concrete entities existing in space and time (persistence)

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary



# Class Diagrams

- 1) the most widely used diagram of UML
- 2) models the static design view of a system
- 3) also useful in modelling business objects
- 4) used for specifying the structure (attributes and operations), interfaces and relationships between classes that form the foundation of system architecture
- 5) primary diagram for generating codes from UML models

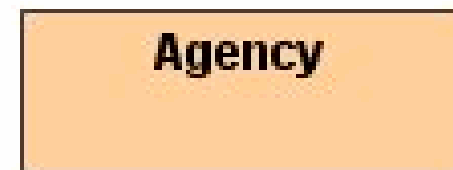
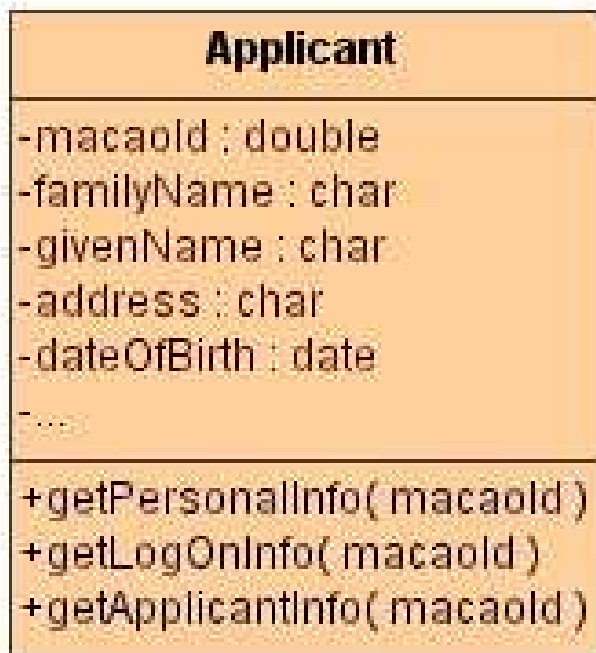
# Class

---

- 1) a description of a set of objects that share the same attributes, operations, relationships and semantics
- 2) a software unit that implements one or more interfaces
- 3) graphically rendered as a rectangle usually including a name, attributes and operations

# Example: Classes

---



# Class Notation

---

Basic notation: a solid-outline rectangle with three compartments separated by horizontal lines.

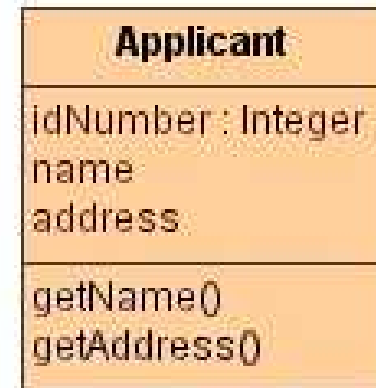
Three compartments:

- 1) the top compartment holds the class name and other general properties of the class
- 2) the middle compartment holds a list of attributes
- 3) the bottom compartment holds a list of operations

Alternative styles for presentation:

- 1) suppress the attributes compartment
- 2) suppress the operation compartment

# Example: Class Notation



# Stereotypes for Classes 1

The list of attributes or operations in a class may be organized using **stereotypes**.

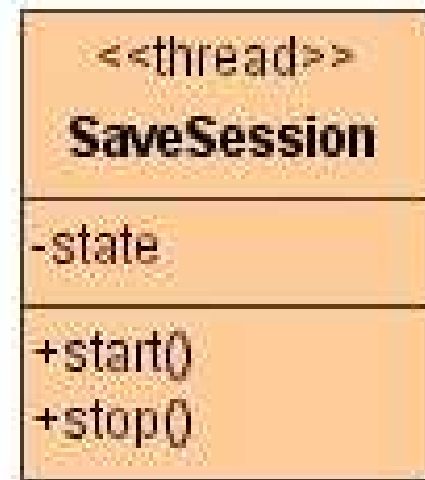
**Stereotype:**

- 1) a new class of a metamodel element introduced at the modelling time
- 2) represents a sub-class of an existing metamodel element with the same form (attributes and operations) but with different intent
- 3) notation – the name of a metamodel element within the matched guillemets e.g.  
<<changeintent>>

# Stereotypes for Classes 2

UML provides a number of stereotypes of classes and data types such as:

- 1) thread
- 2) event
- 3) entity
- 4) process
- 5) utility
- 6) metaclass
- 7) powerclass
- 8) enumeration
- 9) ...



# Example: Stereotypes

---

Stereotype	Description
Thread	An active class which specifies a lightweight flow that can execute concurrently with other threads within the same processes.
Process	An active class which specifies a heavyweight flow that can execute concurrently with other processes.
Control	A class which owns almost no information about itself. It represents a behaviour rather than resources and directs the behaviour of other objects almost having no behaviour of its own.
Entity	A class which represents a resource in the real world. It describes its features and their current condition (their state) and preserves its own integrity regardless of where and when it is used.
Utility	A class whose attributes and operations are all class scoped. That is a class which no instance.
Metaclass	A classifier whose objects are all classes.
Powerclass	A classifier whose objects are the children of a given parent.
Enumeration	A user defined data type that defines a set of values that do not change.



# Modelling Scope 1

---

Different scopes can be specified for class features (attributes and operations):

- 1) a feature appears in each instance of the class (or a classifier generally)
- 2) there is just a single instance of the feature for all instances of a class (classifier)

# Modelling Scope 2

---

## Instance scope:

- each instance holds its own value

## Class scope:

- a single value for all instances of the class

Underlining the feature's name indicates the classifier scope.



# Types of Classes

---

## Abstract:

- cannot have direct instances
- the name is written in italics



## Root:

- cannot be a sub-class





## Leaf:

- cannot be a super-class





# Class Relationships 1

Construct	Description	Syntax
<b>Association</b>	A relationship between two or more classifiers that involves connection among instances.	
<b>Aggregation</b>	A special form of association that specifies a whole-part relationship between the aggregate(whole) and the component part.	

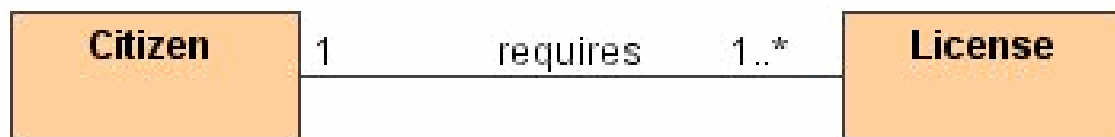
# Class Relationships 2

---

Construct	Description	Syntax
<b>Generalization</b>	A taxonomic relationship between a more general and a more specific element.	
<b>Dependency</b>	A relationship between two modelling elements, in which a change to one modelling element (the independent element) will affect the other modelling element (the dependent element).	

# Multiplicities for Classes

- 1) shows how many objects of one class can be associated with one object of another class
- 2) example: a citizen can apply for one or more licenses, and a license is required by one citizen



# Multiplicities for Attributes

- 1) can specify how many instances of an attribute can be associated with one instance of the class
- 2) example: an applicant will have 2 addresses

Applicant
idNumber : Integer name address [2]
getName() getAddress()

# Multiplicities Syntax

<b><i>Value</i></b>	<b><i>Description</i></b>
0..0	Zero
0..1	Zero or one
0..*	Zero or more
1..1	One
1..*	One or more
*	Unlimited number
<literal>	Exact number (Example: 4)
<literal>..*	Exact number or more (Example: 4..* indicating 4 or more)
<literal>..<literal>	Specified range (Example: 4..13)
<literal>..<literal>, <literal>	Specified range or exact number (Example: 4..13,31 indicating 4 through 13 and 31)
<literal>..<literal>,<literal>..<literal>	Multiple specified ranges (Example: 4..13, 31-41)



# Roles

---

A **role** names a behaviour of an entity participating in a particular relationship.

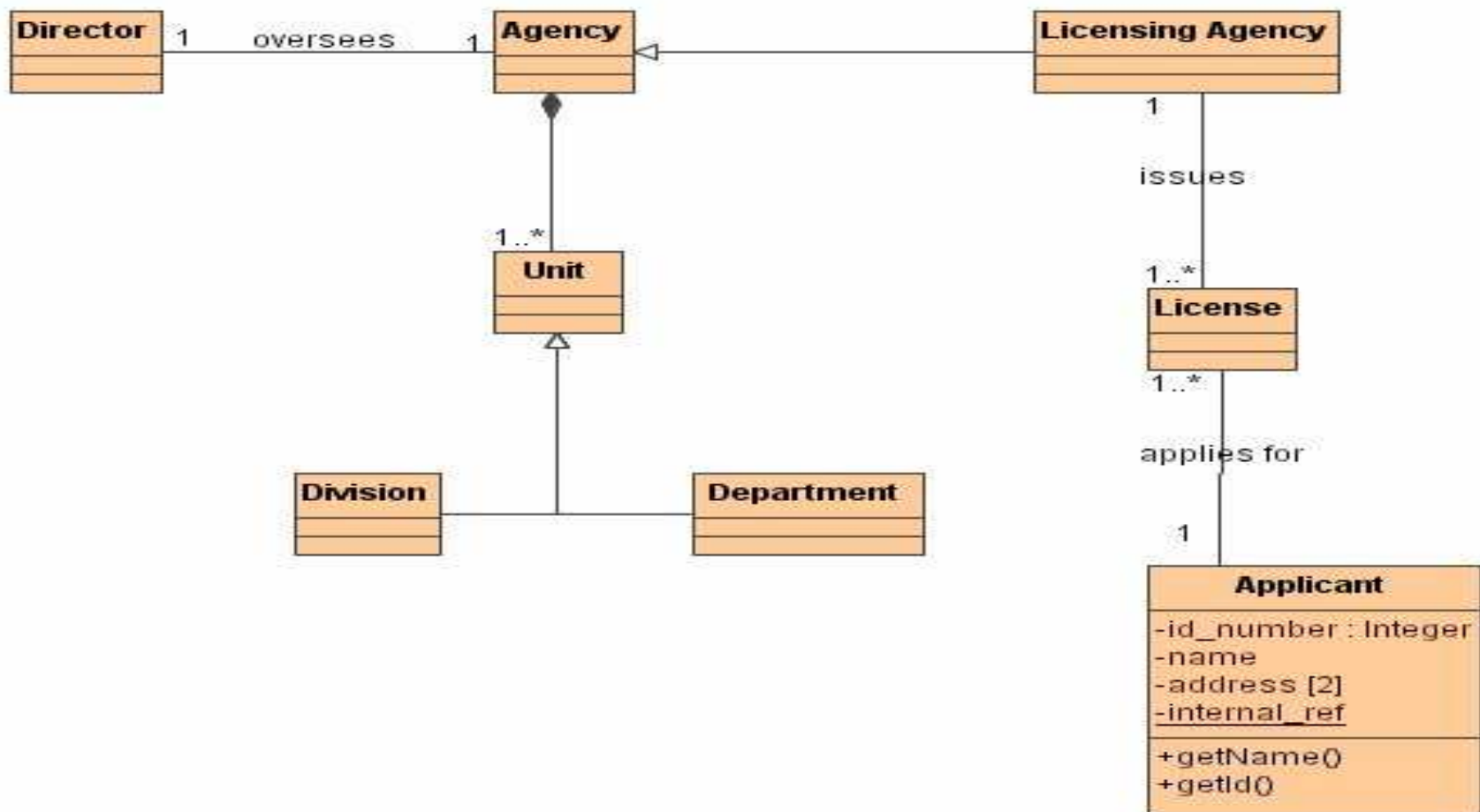
Notation:

- add the name of the role at the end of the association line next to the class icon

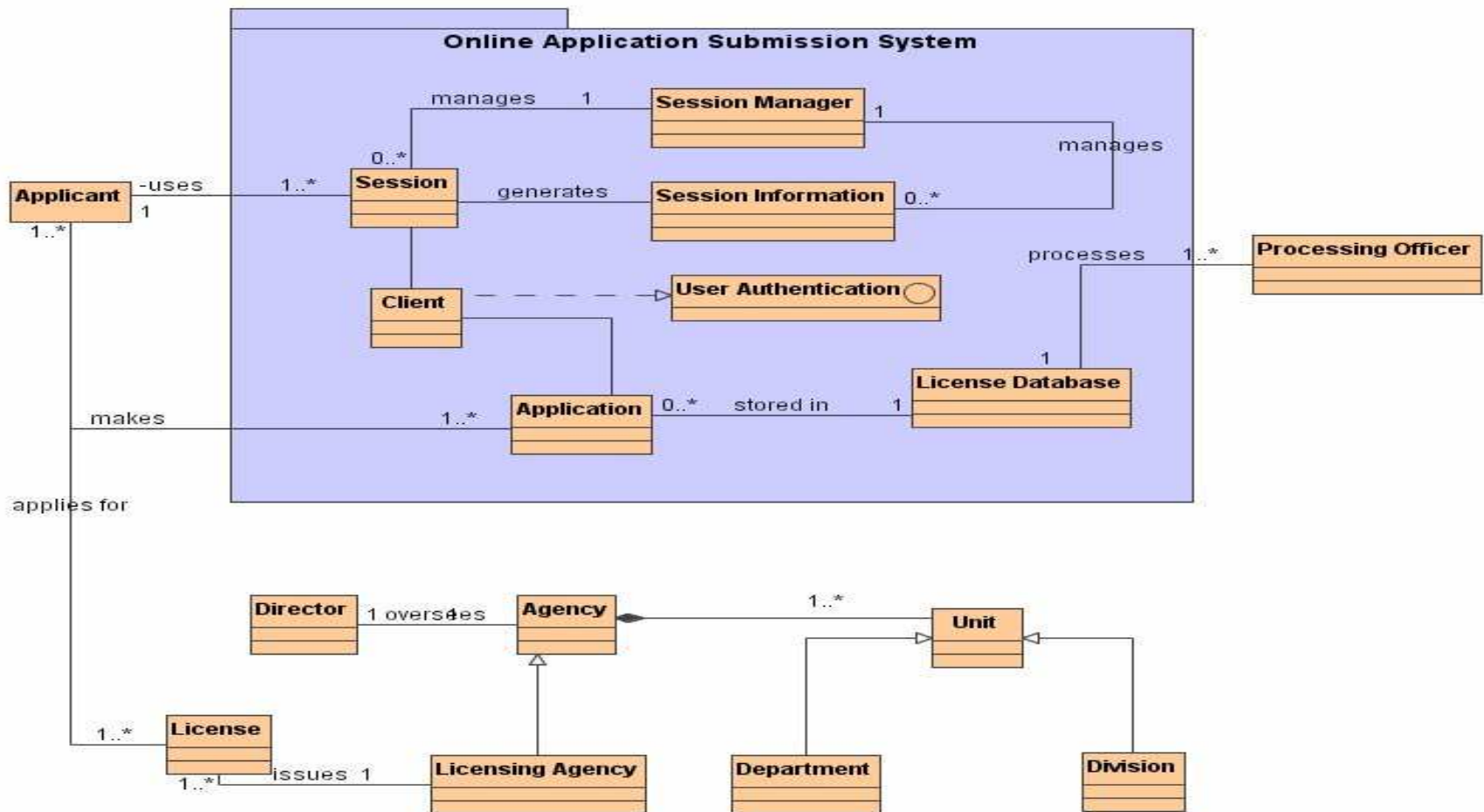
Example: a citizen may play different roles depending on the context, e.g. can owns a car, or be an employee, or ...



# Example: Class Diagrams 1



# Example: Class Diagrams 2



# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Object Diagram 1

---

- 1) models the instances of classes contained in class diagrams
- 2) shows a set of objects and their relationships at a point in time
- 3) modelling object structures involves taking a snapshot of a system at a given moment in time
- 4) is an instance of a class diagram or the static part of an interaction diagram

# Object Diagram 2

---

5) is used to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships

6) contains:

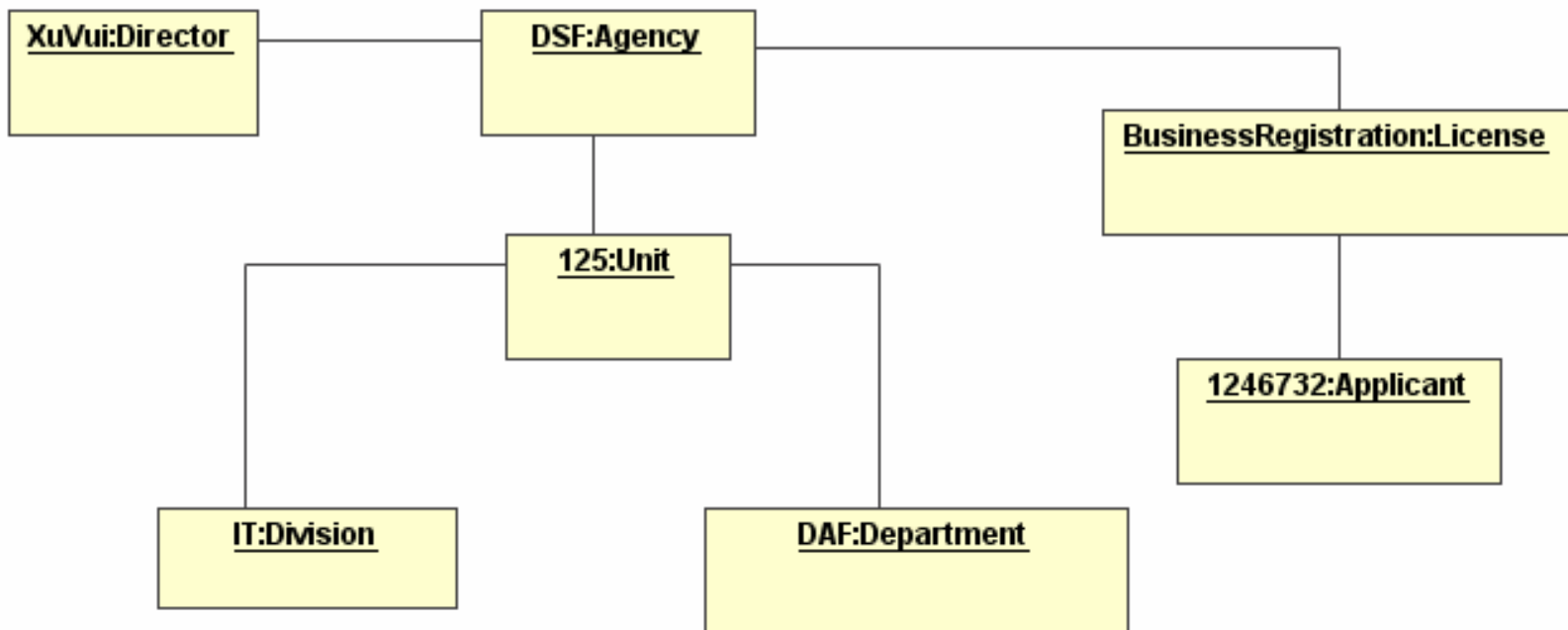
a) **objects**

b) **links**

# Creating an Object Diagram

- 1) identify the function or behaviour you want to model that results from the interaction of a set of classes, interfaces and other artifacts
- 2) for each function or behaviour, identify the artifacts that participate in the collaboration as well as their relationships
- 3) consider one scenario that invokes the function or behaviour. Freeze the scenario and render each participating object
- 4) expose the state and attribute values of each object, as necessary to understand the scenario
- 5) expose the links among these objects

# Example: Object Diagram





# Requirements Modelling: Behavioural Modelling

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Behavioural Diagrams 1

- 1) represent how objects behave when you put them to work using the structure already defined in structural diagrams
- 2) model how the objects communicate in order to accomplish system tasks
- 3) describe how the system:
  - a) responds to actions from the users
  - b) maintains internal integrity
  - c) moves data
  - d) creates and manipulates objects, ...

# Behavioural Diagrams 2

- 4) describe discrete pieces of the system, such as individual **scenarios** or operations

## Note:

no need to specify behavioural diagrams for all system behaviours as simple behaviours may not need a visual explanation of the communication required to accomplish them

# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Scenarios

---

## Definition

**Scenario** is a textual description of how a system behaves under a specific set of circumstances.

The behaviour described in the use cases is the basis for building scenarios.

Scenarios also provide a basis for developing test cases and acceptance-level test plans.

# Example: Scenarios 1

- 1) consider the use case “Track License Application”
- 2) there are at least two possible scenarios:
  - a) the applicant enters the license application number; the system retrieves the information related to it; and the system displays this information
  - b) the applicant enters the license application number but this number does not exist in the agency’s database, in which case the system displays an error message

# Example: Scenarios 2

**Scenario:** the applicant enters the license application number; the system retrieves the information related to it; and the system displays this information

**Steps:**

- 1) Applicant requests to track status of license application
- 2) System displays the log on form
- 3) Applicant enters log on information
- 4) Applicant submits log on information
- 5) System validates applicant
- 6) System displays form to enter the tracking number
- 7) Applicant enters the tracking number
- 8) Applicant submits license number
- 9) System retrieves license information
- 10) System displays license information



# Sequence Diagrams

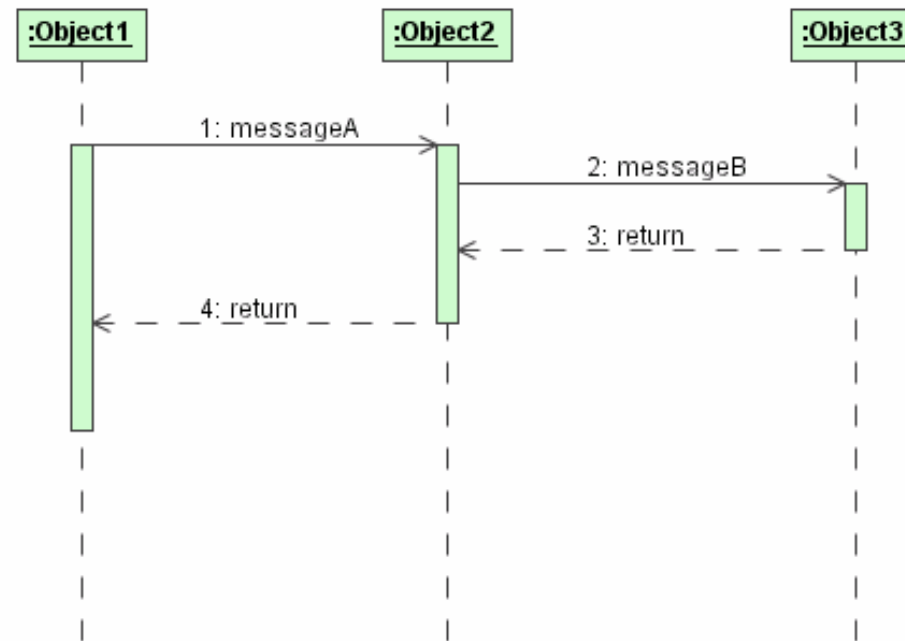
---

- 1) show how objects interact by sending messages among them in order to provide a specific behaviour
- 2) address the dynamic behaviour of a system with special emphasis on the chronological ordering of messages
- 3) show a set of messages arranged in time sequence
- 4) are used to show the behaviour sequence of a use case

# Sequence Diagram - Elements

Two elements are used to build sequence diagrams:

- 1) **object lifelines**
- 2) **messages** or **stimuli**



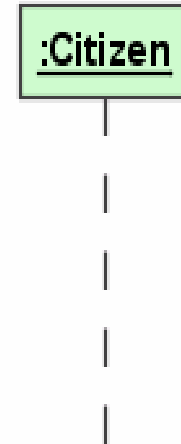
# Object Lifeline

---

The notation combines the **object icon** and a **timeline**.

The timeline is a line that runs from the beginning of a scenario at the top of the diagram to the end of the scenario at the bottom of the diagram.

If an object is created and destructed during the messages sequence, then the lifeline represents the whole lifecycle of the object.



# Message

---

## Definition

**Message** is a description of some type of communication between objects.

It is a unit of communication between objects.

The sender object may invoke an operation, raise a signal or cause the creation or destruction of the target object.

Notation: is modelled as an arrow where the tail of the arrow identifies the sender and the head points to the receiver.

sender            receiver

# Stimulus

---

## Definition

**Stimulus** is an item of communication between two objects, and has the following characteristics:

- 1) it is associated with both a sending and a receiving object
- 2) it travels across a link
- 3) it may invoke an operation, raise a signal (asynchronous message), create or destroy an object
- 4) it may include parameters/arguments in the form of either primitive values or object references
- 5) it is associated with the procedure that causes it to be sent

# Example: Stimulus

- 1) **Example 1 – invoking an operation**: The citizen object submits the application form by sending a message to the system. The system receives the message and executes the associated procedure.
- 2) **Example 2 – create an object**: when the System receives the message from the applicant submitting the application form, it creates an object to support the new session.

# Messages and Stimuli

A **message** is the formal **specification** of a **stimulus**.

A stimulus is an instance of a message.

The specification includes:

- 1) the roles that the sender object and the receiver object play in the interaction
- 2) the procedure that dispatches the stimulus

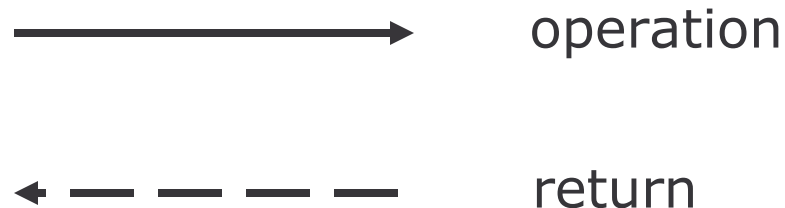
For each message we need to define:

- 1) the name of the invoked operation and its parameters
- 2) the information returned from the message.

# Operations and Returns

The **operation** specifies the procedure that the message invokes on the receiving object.

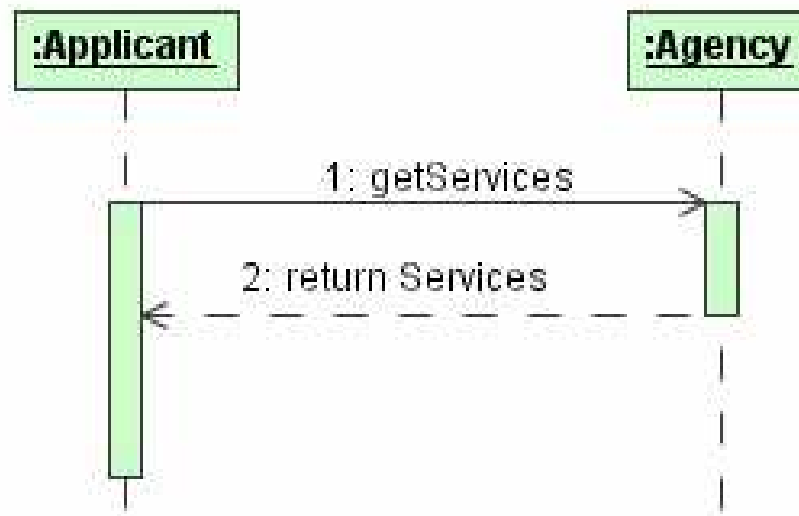
The **return** contains the information passed back from the receiver to the sender. An empty return is valid.





# Example: Operations - Returns

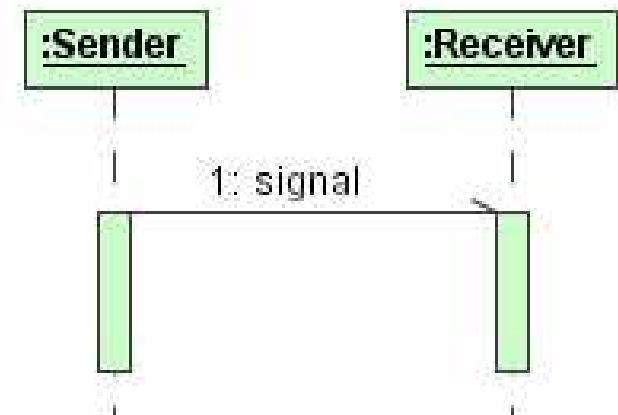
Example: The *Applicant* object sends a message to the *Agency* object to get the information of which are the different services its provides. The *Agency* object returns this information.



# Signals

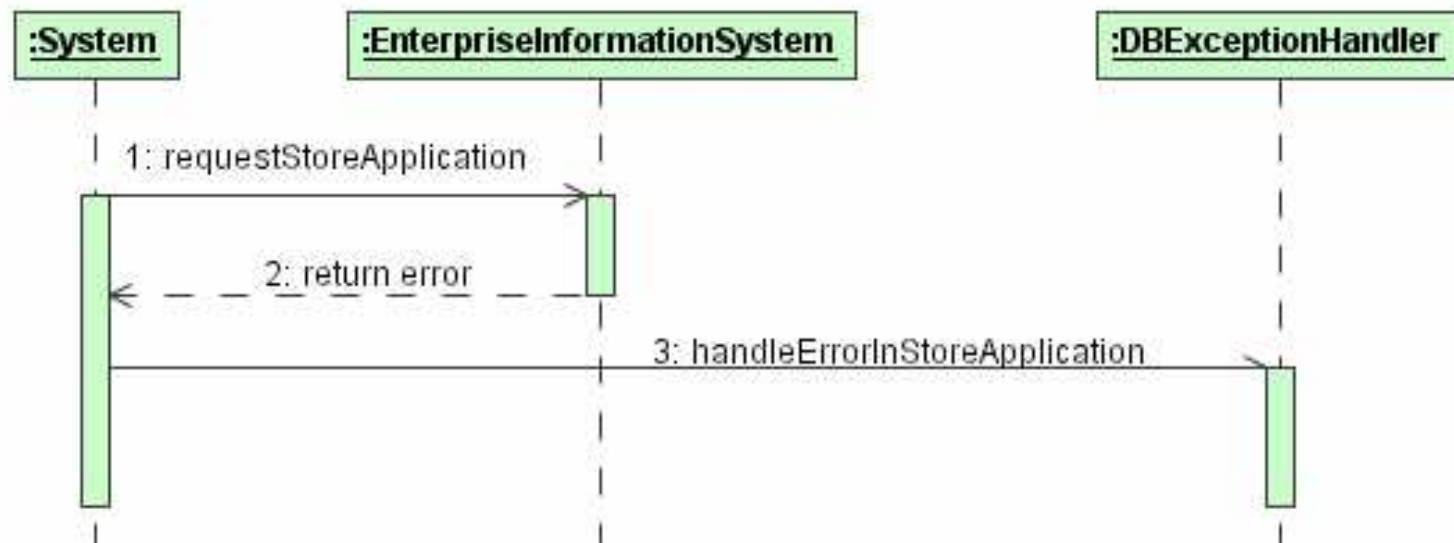
---

- 1) an object may raise a signal through a message
- 2) a **signal** is a special type of a class associated with an event that can trigger a procedure within the receiving object
- 3) a signal does not require a return from the receiving object
- 4) an exception is a special type of a signal
- 5) **throwing an exception** means sending out a message containing an object that describes the error condition



# Example: Signal

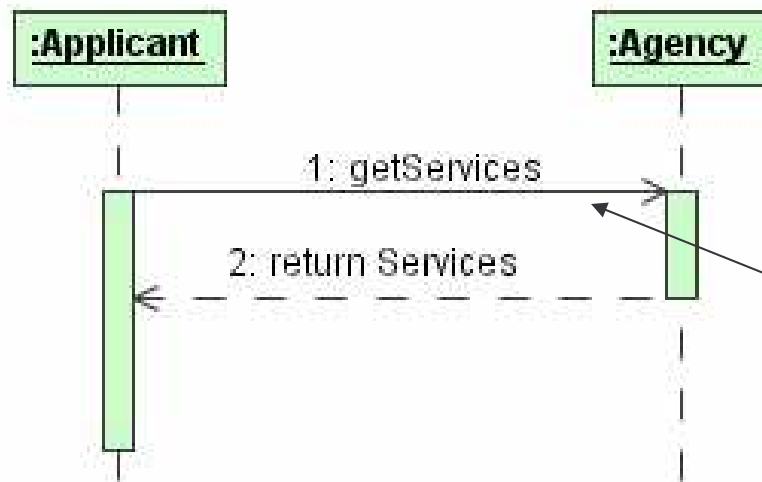
Each time the System object receives a message indicating that some abnormal situation occurred with the database, it raises a signal to the object DBExceptionHandler for the complete treatment of the error.



# Identification of Messages

A message number or a message name is used to properly identify messages.

Example:



In this example, we identify the message **getServices** with **1**.

# Example: Message Syntax

Example:

2.1, 2.5, 6 / 8: **getCitizenAddress** \* [foreach ApplicationForm] *return* text  
:= getCitizenPersonalAddress(Citizen.Id:Integer)

In this example, we are specifying message number 8 called **getCitizenAddress**.

This message will be executed more than once (\*), one time for each ApplicationForm.

Each message will call the operation getCitizenPersonalAddress of the receiving object, sending as parameter the CitizenId that is of type Integer, and will return a value of type text.

For the execution of this message, it is required that messages 2.1, 2.5 and 6 have already been executed.

# Message Syntax 1

*predecessors '/' sequence-term iteration [condition] return ':=' operation*

where:

- 1) *predecessors* refers to a list of comma-separated sequence numbers of messages that must come before the current message
  - predecessors on the diagram are assumed so they don't need to be included
- 2) *sequence-term* may be either a number or a name that identifies the message

# Message Syntax 2

---

- 3) *iteration* refers to the need to execute one or more messages in a sequence more than once
  - 3) one message: add an iteration symbol (\*) and a condition to control the number of iterations
  - 4) many messages: enclose the set of messages in a box
- 4) *condition* is used to specify the control of the iteration and is expressed as a text enclosed within square brackets
- 5) *return* may include a list of values sent back to the sender by the receiver
- 6) *operation* defines the name of the operation and optionally its parameters and a return value

# Case Study Model Example 1

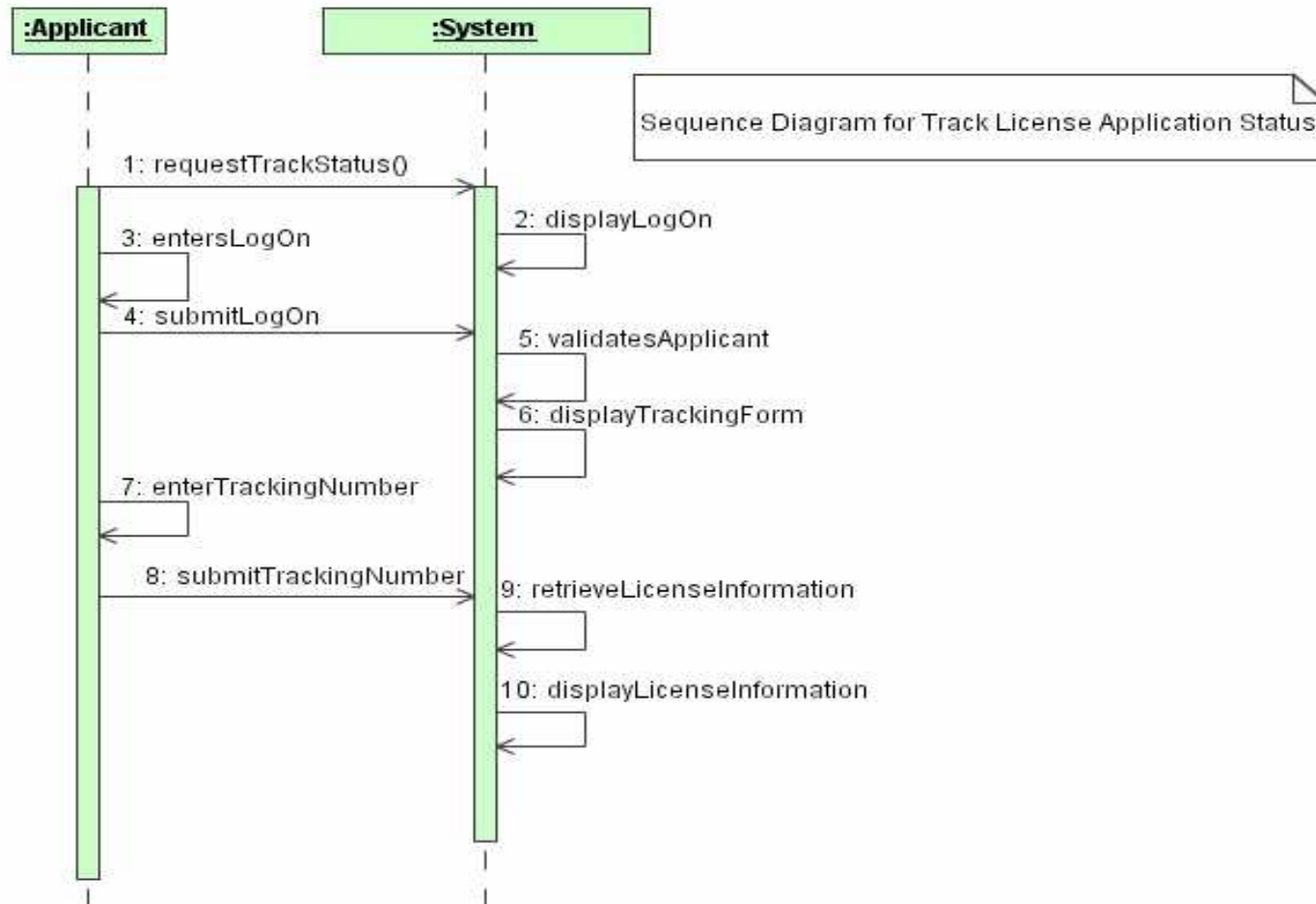
Recall the example explained for scenarios: an applicant tracks the status of a license already applied through the system and the system displays the license information.

## Procedure:

- 1) Applicant requests to track status of license application
- 2) System displays the log on form
- 3) Applicant enters log on information
- 4) Applicant submits log on information
- 5) System validates applicant
- 6) System displays form to enter the tracking number
- 7) Applicant enters the application the tracking number
- 8) Applicant submits tracking number
- 9) System retrieves license information
- 10) System displays license information



# Case Study Model Example 2



# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Statechart Diagrams

---

Statechart diagrams define a notation for describing state machines.

**State machines** capture the changes in an object throughout its lifecycle as they occur in response to external events.

The statechart diagram identifies both the external events and internal events that can change the object's state.

The scope of a statechart is the entire life of one object.

The foundation of statecharts is the relationship between **states** and **events**.

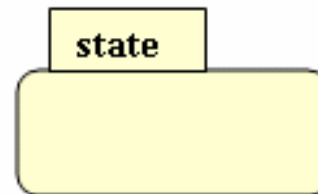
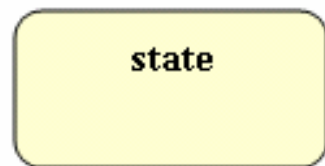
# States

---

## Definition

**State** is the current condition of an object reflected by the values of its attributes and its links to other objects.

Notation:



# Initial State

---

The **initial state** identifies or points to the state in which an object is created or constructed.

The initial state is called a **pseudo-state** because it does not really have the features of an actual state, but it helps clarify the purpose of another state of the diagram.

Notation:



# Final State

---

The **final state** is the state in which once reached, an object can never do a transition to another state.

Also, the final state may mean that the object has actually been destroyed and can no longer be accessed.

Notation:



# Events

---

## Definition

**Event** is an occurrence of a stimulus that can trigger a state transition.

An event may be:

- 1) the **receipt of a signal**, e.g. the reception of an exception or a notice to cancel an action
- 2) the **receipt of a call**, that is the invocation of an operation, e.g. for changing the expiration date of a license or for printing approved licenses.

An event on a statechart diagram corresponds to a message on a sequence diagram.

Notation:



# Example: Statechart Diagram





# Special Events

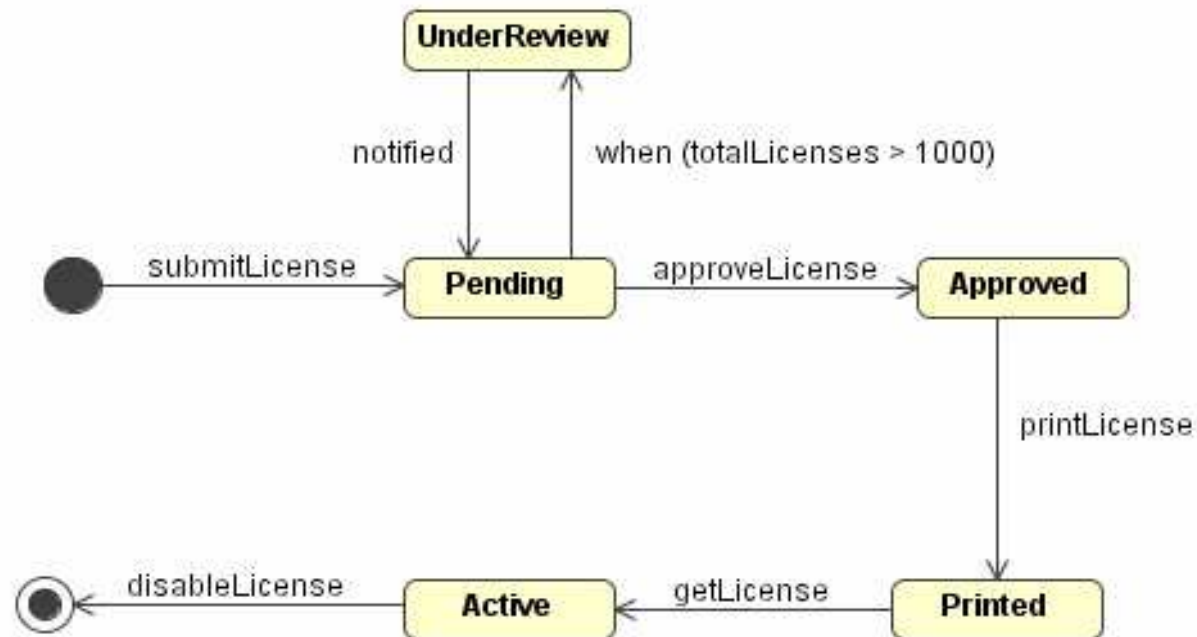
---

An event may also be the recognition of some condition in the environment or within the object itself, such as:

- 1) **change event**: A predefined condition becoming true. Example: when the amount of submitted licenses by day is over a predefined quantity, an action such as sending a notice is required.
- 2) **elapsed-time events**: The passage of a designated period of time. Example: for printed licenses not reclaimed by anybody after some period, some action is required to destroy those licenses.

# Example: Change Event

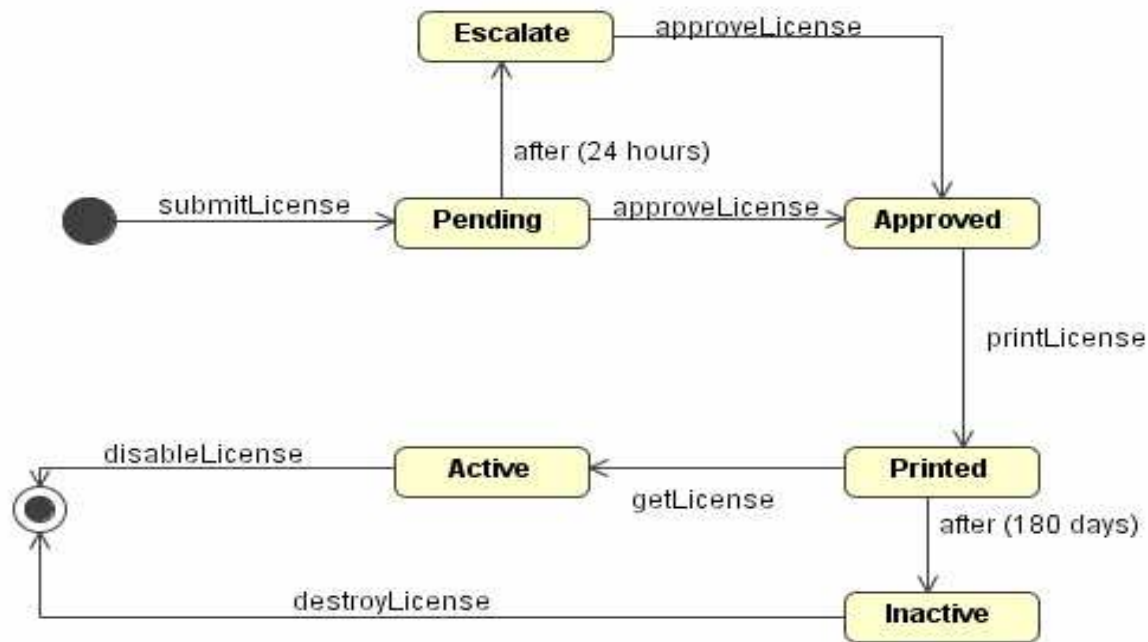
A **change event** is an event where the transition is based on a change in the object or a change in the environment.



The event-name is the keyword **when** followed by a boolean expression enclosed in parenthesis.

# Example: Elapsed-Time Event

An **elapsed-time** event is an event where the transition is triggered because of the passage of time.



The event-name is the keyword “**after**” followed by a numerical expression enclosed in parenthesis that represents the amount of time.

# Event syntax 1

---

Example:

```
approveLicense(License.Id) [requirements=ok] /  
setExistLicense(true)
```

where:

- 1) the **event name** is approveLicense
- 2) the **event parameter** is License.Id
- 3) the **guard condition** specifies that *requirements* (attribute of the object) should be *ok*. This condition determines whether the receiving object should respond to the event or not
- 4) the **action** executed in the receiving object is a call to the method setExistLicense sending true as parameter

# Event syntax 2

---

event-name '(' [comma-separated-parameters-list] ')' '  
['[guard-condition]'] / [action-expression]

where:

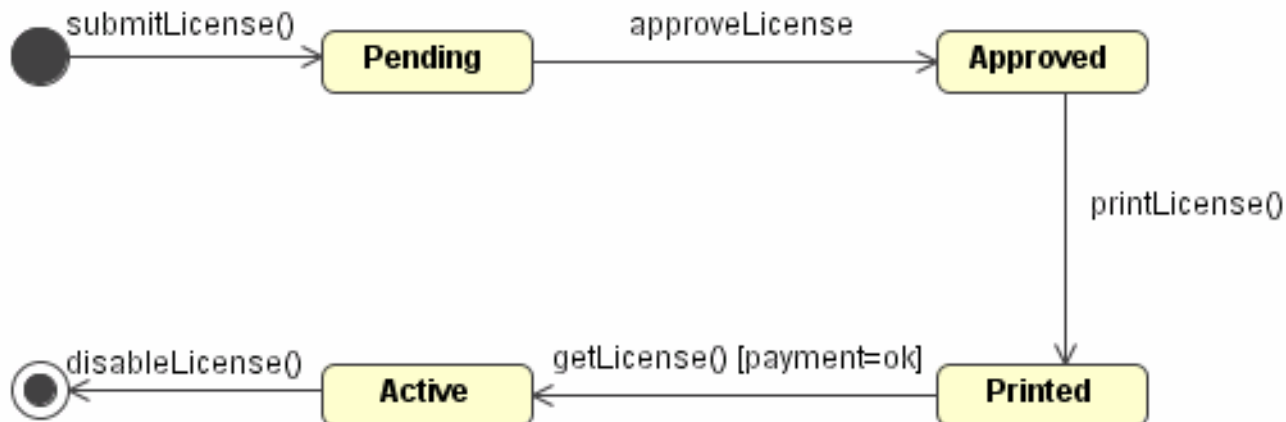
- 1) **event-name** identifies the event
- 2) **parameters-list** defines the data values that are passed with the event for use by the receiving object in its response to the event
- 3) **guard-condition** determines whether the receiving object should respond to the event
- 4) **action-expression** defines how the receiving object must respond to the event

# Guard Condition

---

Typically, an event is received and responded to unconditionally.

However, the receipt of an event may be conditional; the test needed is called the **guard condition**.



# Event Actions

---

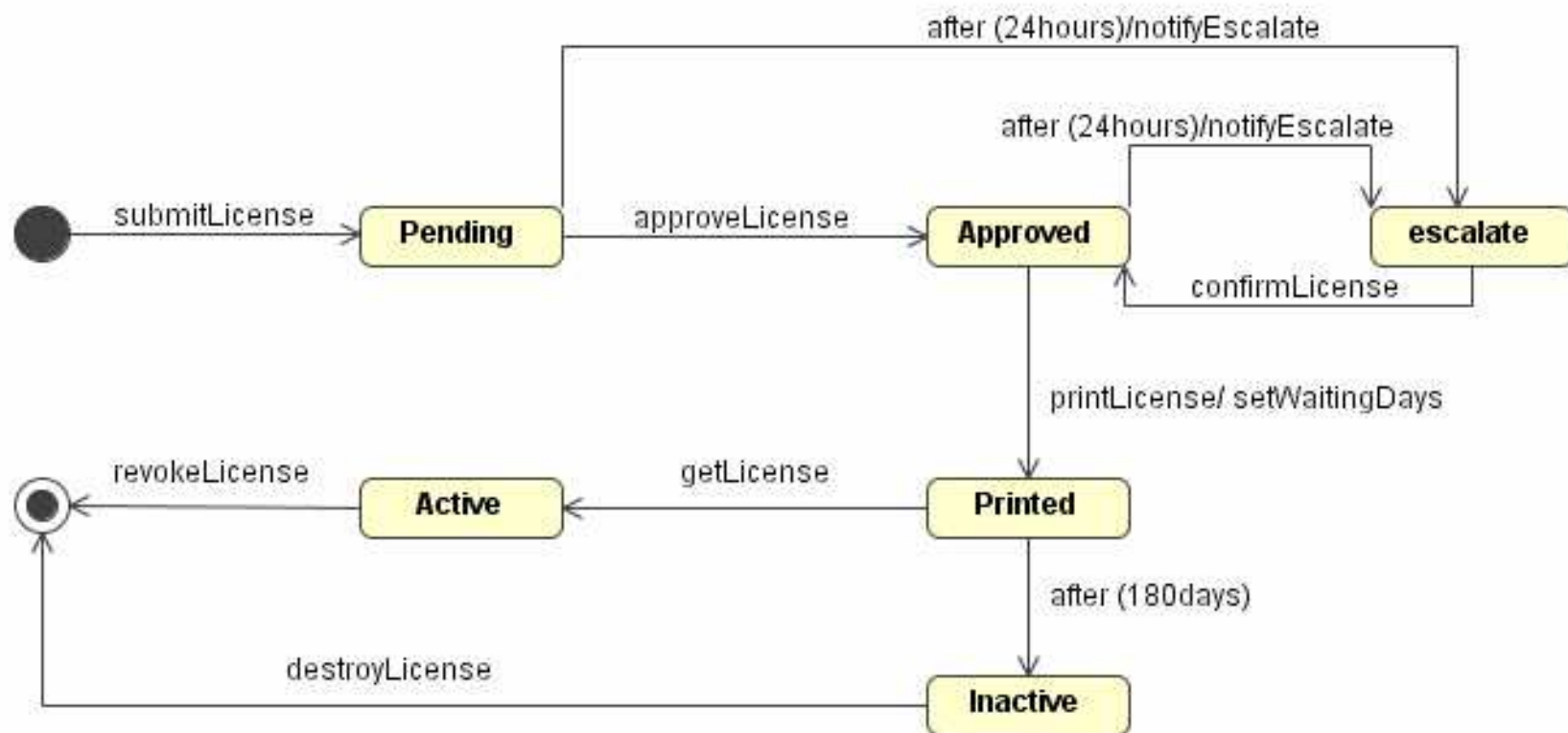
The response to an event has to explain how to change the attribute values that define the object's state.

The behaviour associated with an event is called an **action expression**.

An action expression is part of a transition event, it is a part of the change from one state to another.

An action expression is an atomic model of execution, and is referred to as run-to-completion semantics.

# Example: Event Actions





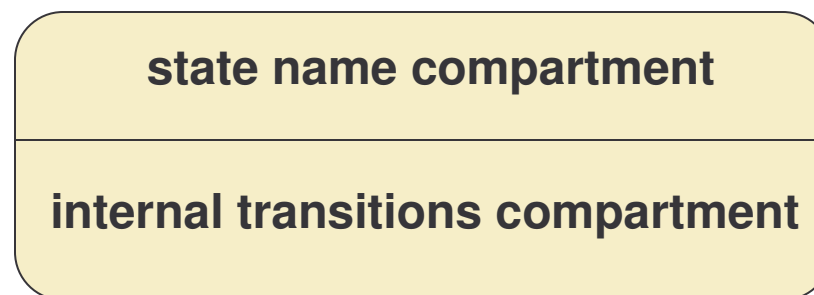
# Internal Compartment

---

The state icon can be expanded to model what the object can do while it is in a given state.

The notation splits the state icon into two compartments, the **name compartment** and the **internal transitions compartment**.

The internal transitions compartments contains information about actions, activities and internal transitions specific to that state.



# Entry Actions

---

More than one event can change the object to the same state.

When the same action takes place in **all events** that goes into the state, it is possible to write the action only once as an **entry action**.

Notation:

- 1) use the keyword **entry** followed by a slash and the action or actions that need to be performed every time you enter the state
- 2) entry actions are placed in the internal transitions compartment.

# Exit Actions

---

The same simplification can be used for actions associated with events that trigger a transition out of a state.

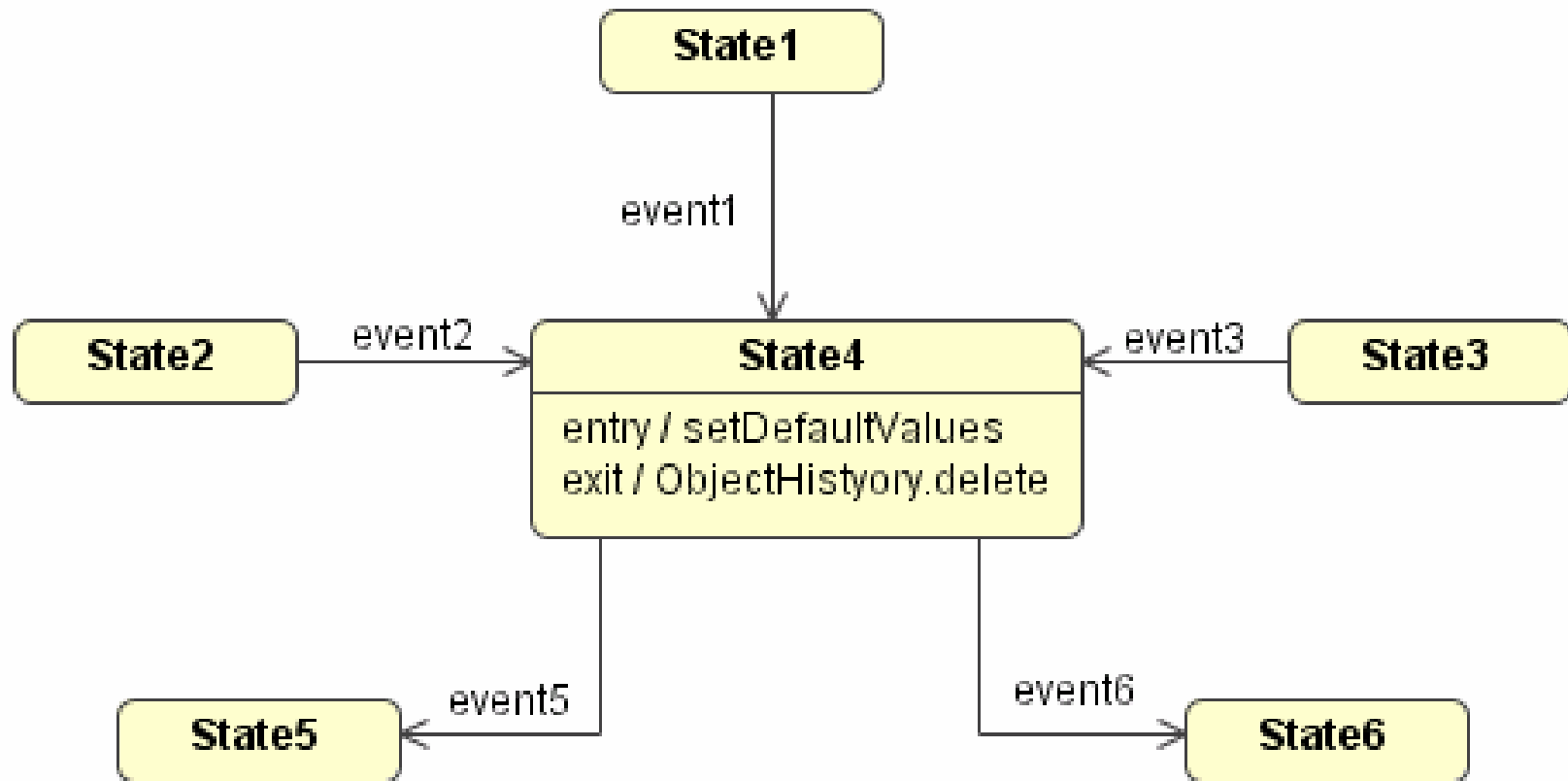
They are called **exit actions**.

Notation:

- 1) use the keyword **exit** followed by a slash and the action or actions that are performed every time you exit the state
- 2) exit actions are placed in the internal transitions compartment.

It may only be used when the action takes places **every time** you exit the state.

# Example: Entry / Exit Actions



# Activities 1

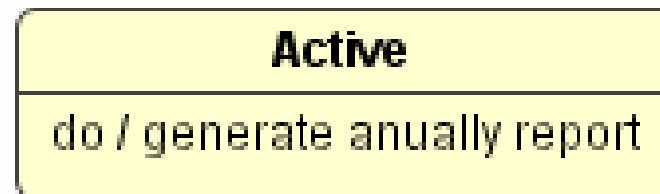
---

Activities are **processes** performed within a state.

Activities may be interrupted because they do not affect the state of the object.

Notation:

- 1) use the keyword **do** followed by a slash and one or more activities
- 2) activities are placed in the internal transitions compartment.



# Activities 2

---

An activity should be performed from the time the object enters the state until either the object leaves the state or they finish.

If an event produces a transition out of the activity state, the object must shut down properly and exit the state.

# Internal Transitions

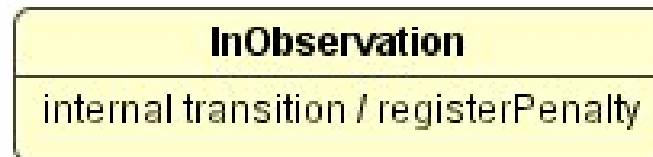
---

An event that can be handled completely without a change in state is called an **internal transition**.

It also can specify guard conditions and actions.

Notation:

- 1) uses the keyword **internal transition** followed by a slash and one event action
- 2) they are placed in the internal transitions compartment



# Order of Events

---

- 1) if an activity is in process in the current state, interrupt it and finish it in a proper way
- 2) execute the exit action(s)
- 3) execute the actions associated with the event that triggered the transition
- 4) execute the entry action(s) of the new state
- 5) begin executing the activity or activities of the new state.



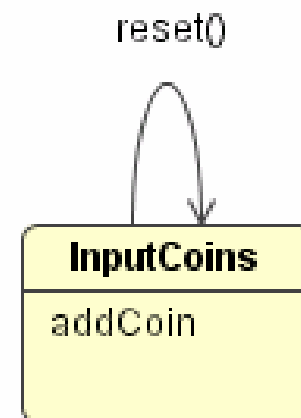
# Self Transition

---

A self-transition is an event that is sufficiently significant to interrupt what the object is doing.

It forces the object to exit the current state and return to the same state.

The result is to stop any activity within the object, exit the current state and re-enter the state.



# Important Features

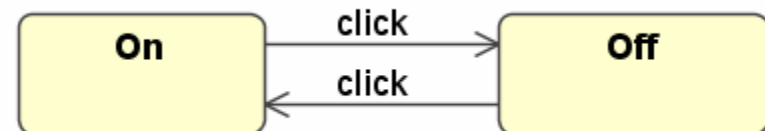
Within the statechart diagram:

- 1) an object need not know who sent the message
- 2) an object is only responsible for how it responds to the event

Focusing on the condition of the object and how it responds to the events, which object sends the message becomes irrelevant and the model is simplified

The state of the object when it receives an event can affect the object's response

event + state = response



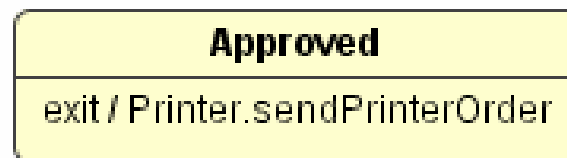
# Defining Send Events

---

Sometimes the object modelled by the statechart needs to send a message to another object, in this case the outgoing event must define which is the receiving object. Also, this event must be caught by the receiving object.

A message send to another object is called a **send event**.

Notation: provide the object name before the action expression with a period separating both.



# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary

# Relating Diagrams 1

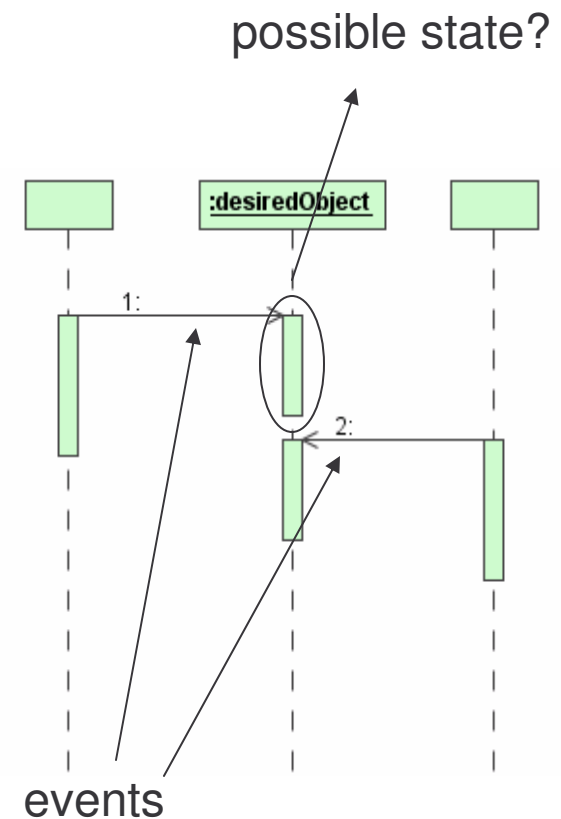
- 1) the sequence diagram models the interactions between objects
- 2) the statechart diagram models the effect that these interactions have on the internal structure of each object
- 3) the messages modelled in the sequence diagrams are the external events that place demands on objects

# Relating Diagrams 2

- 4) the objects internal responses to those events that cause changes to the objects' states are represented in the statechart diagram
- 5) not all objects need to be modelled with a statechart diagram
- 6) the objects that appear in many interactions and are target of many events are good candidates to be modelled with a statechart diagram

# Deriving Statechart Diagrams

- 1) identify the events directed at the lifeline of the desired object
- 2) identify candidate states by isolating the portions of the lifeline between the incoming events
- 3) name the candidate states using adjectives that describe the condition of the object during the period of time represented by the gap
- 4) add the new state and events to the statechart diagram



# Requirements Modelling

---

## 1) Software Requirements

- a) Problems
- b) Process
- c) Types

## 2) Use Case Modelling:

- a) Concepts
- b) Use Case Diagrams
- c) Templates

## 3) Conceptual Modelling:

- a) Concepts
- b) Class Diagram
- c) Object Diagram

## 4) Behavioural Modelling:

- a) Behavioural Diagrams
- b) Sequence Diagrams
- c) Statechart Diagrams
- d) Relation between them

## 5) Summary



# Summary 1

---

A requirement is a function that a system must perform or a desirable characteristic of a system.

There are different kind of requirements such as functional and non-functional.

Most project failures can be traced back to errors made in requirements gathering and specification.

# Summary 2

---

Use cases are descriptions of a set of action sequences that a system performs to obtain an observable result to the environment.

Use cases may be related using the generalization, include and extend relationships.

Actors are entities that interact with the system causing to respond to business events.

Use case diagrams shows a set of use cases, actors and their relationships.

# Summary 3

---

Conceptual modelling helps in understanding application domains.

Classes are equivalent to Concepts in UML.

Conceptual Class diagrams describe (showing some important attributes of the classes, but no method) and relate the concepts of a domain.

Object diagrams model the instances of classes contained in class diagrams.

# Summary 4

---

Behavioural modelling specifies how objects work together to provide a specific behaviour using their structure.

Sequence diagrams show how objects interact during time in order to deliver a discrete piece of the system functionality.

Statechart diagrams capture the changes in an object or a set of related objects as they occur in response to events.

# Exercise: Requirements 1

## **Section A: Class Exercise**

Consider the Online Licensing Service case study presented earlier.

### *1) Use Case Modelling:*

- a) Using the use case diagram presented in slide 177, provide a detailed use case specification for any two of the use cases shown in the diagram.

# Exercise: Requirements 2

## **Section A: Class Exercise**

### *2) Conceptual Modelling using Class Diagrams:*

- a) List five different concepts (not included in the diagram presented in slide 202 and 203) related to this domain.
- b) Describe the attributes of the classes that represent these five concepts.
- c) Provide a simple conceptual model using a class diagram which relates these concepts with others already presented in slide 202.
- d) Present an object diagram based on the class diagram provided in the previous question specifying the objects states.

# Exercise: Requirements 3

---

## Section A: Class Exercise

### *3) Behavioural Modelling using Sequence and Statechart Diagrams:*

- a) Describe two scenarios for the use case detailed in point one of this exercise.
- b) Present the sequence diagrams for these scenarios.
- c) Select an object that has a relevant behaviour and provide the statechart diagram to model the different states of its lifecycle. If possible, try to include an activity, a change-event and an elapsed-time event.

# Exercise: Requirements 4

## **Section B: Project**

- 4) List the core functional requirements of the system you wish to develop. For each requirement listed, provide the requirement identifier, description, and cross references.
- 5) List five non functional requirements of the system using the format specified in question 4.
- 6) Identify the use cases for your system.
- 7) Identify the actors related to each of the use cases identified in question 6.



# Exercise: Requirements 5

## **Section B: Project**

- 8) Provide a detailed specification for the uses cases listed in question 6 using the template provided in slide 173.
- 9) Provide a use case model to show the relationships between actors and user cases for your system.
- 10) Provide a glossary that describes the core concepts of your application domain (at least 10 concepts). You may suggest any glossary format of your choice.
- 11) Use a class diagram to relate these concepts identified in question 10. In addition, update the glossary to define the meanings of the relationships used in this diagram.

# Exercise: Requirements 6

## **Section B: Project**

- 12) Identify the scenarios with relevant behaviour.
- 13) Provide sequence diagrams for different scenarios of the use cases mentioned in the previous question.
- 14) Identify objects which may present different states.
- 15) Provide statecharts for the objects mentioned previously.

# Architecture Modelling

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# Software Architecture Concepts

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary

# Software Architecture 1

---

## Definition

An **architecture** is:

1. a set of significant decisions about the organization of a software system
2. the selection of structural elements and their interfaces by which the system is composed together with a behaviour as specified in the collaborations amongst elements
3. the composition of these structural and behavioural element into progressively larger subsystem
4. the architectural style that guide this organization – the elements and their interfaces, their collaboration and composition

# Software Architecture 2

---

## Architecture involves:

- 1) structural organization of a system from its components,
- 2) ensemble of **elements** that collaborate to constitute the system
- 3) how elements interact to provide the system's overall behaviour or required functionality

## Architectural concerns:

- 1) structural or static
- 2) behavioural

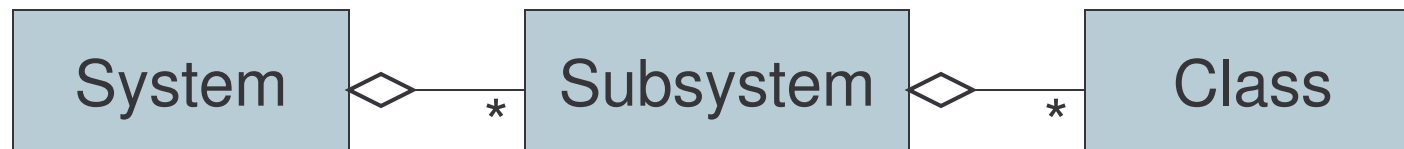


# Subsystems and Classes

A solution domain may be decomposed into smaller parts called subsystems.

Subsystems are composed of solution domain classes (design classes).

Subsystems may be recursively decomposed into simpler subsystems.



# Subsystems and Services 1

A subsystem is characterized by the services it provides to other subsystems.

A service is a set of related operations that share a common purpose.

## Example of a service:

notification service involves the following operations –  
send notices, lookup notifications channels, subscribe  
and un-subscribe to a channel

The set of operations of a subsystem are available to other subsystem through the subsystem's interface.

# Subsystems and Services 2

Subsystem interface or API includes:

- 1) name of the operations
- 2) parameters
- 3) high-level behaviour
- 4) return values

The notion of services allow to focus more on the interfaces rather than the implementation to minimize the impact of change.

# Coupling

---

## Definition

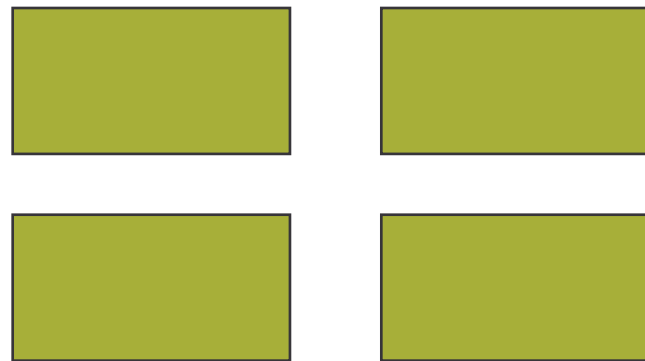
Coupling is the strength of dependencies between two sub-systems.

Subsystem independence allows for easy understanding, modification and maintenance.

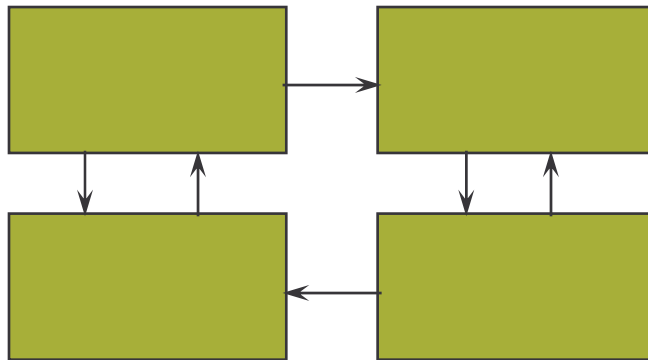
Loose coupling minimizes the impact of one subsystem on others.

A desirable property of subsystem decomposition is that the resulting subsystems be loosely coupled.

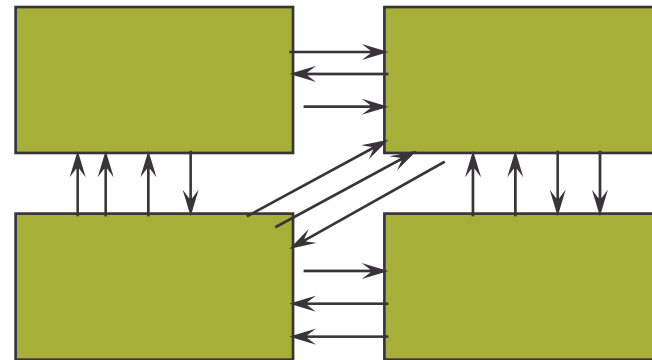
# Coupling: Example



Uncoupled -  
no dependencies



Loosely coupled -  
some dependencies



Highly coupled -  
many dependencies

# Cohesion or Coherence

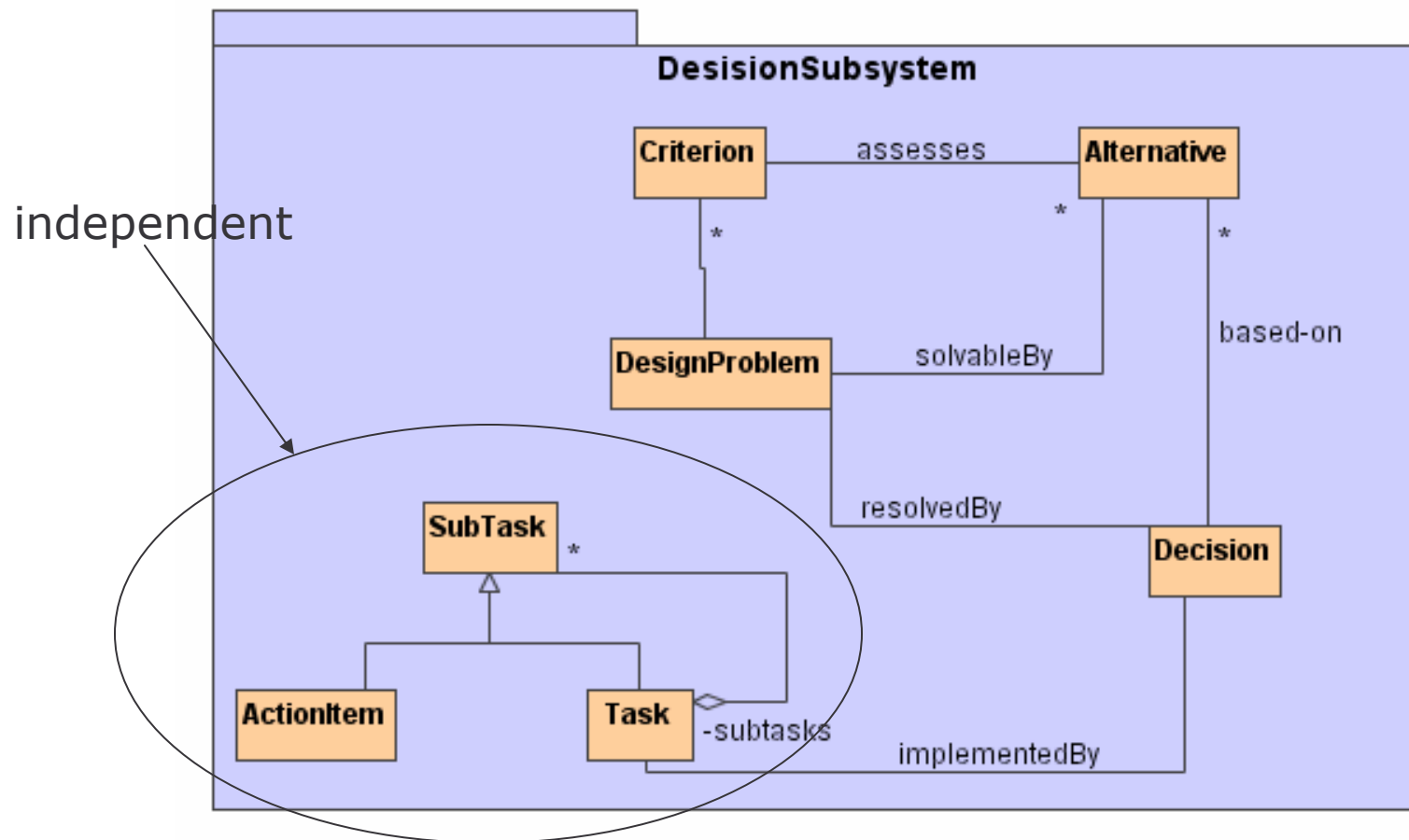
---

## Definition

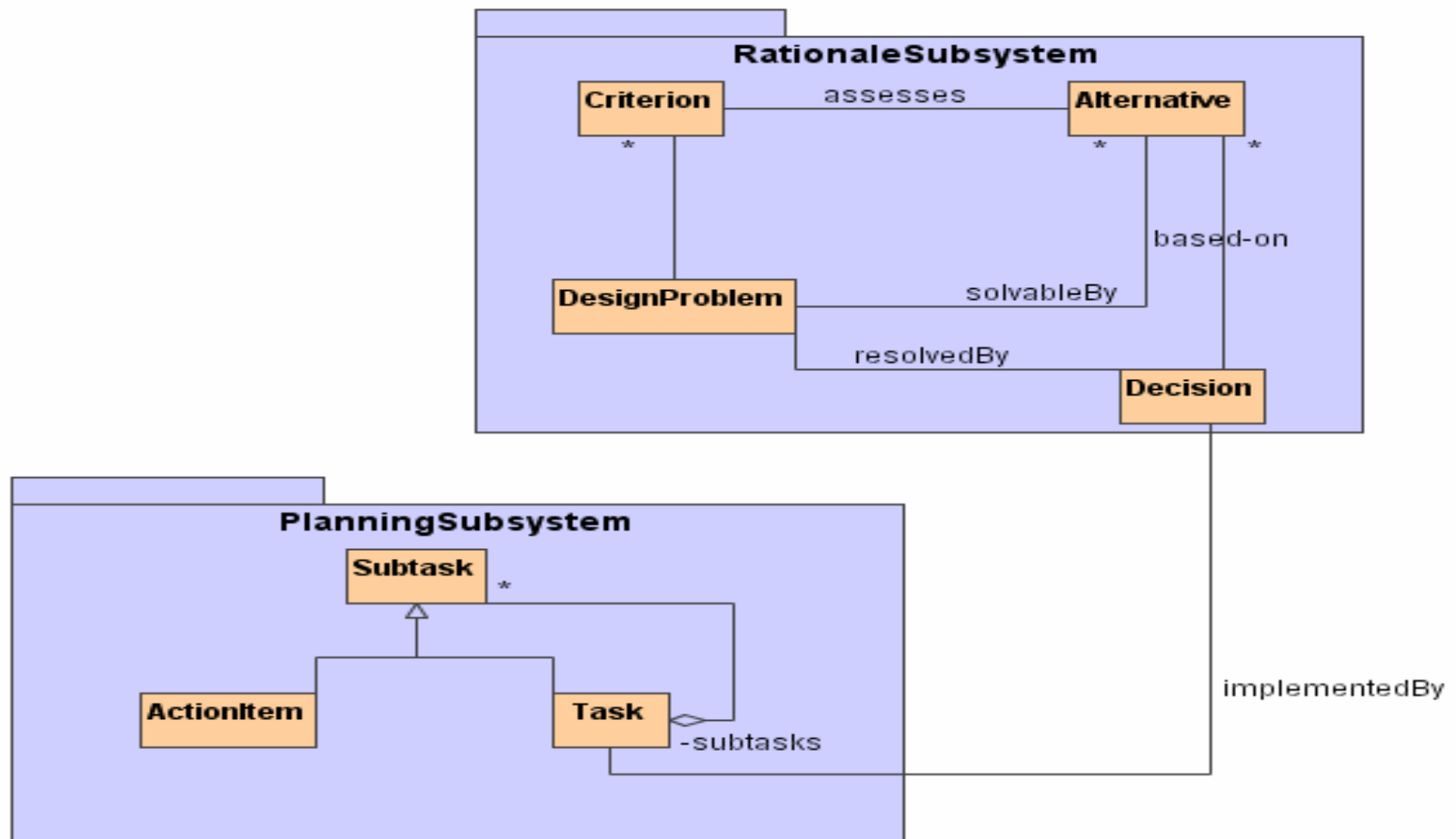
**Coherence** or cohesion is the strength of dependencies within a subsystem.

- 1) the internal “glue” with which a subsystem is constructed
- 2) a component is cohesive if all its elements are directed toward a task and the elements are essential for performing the same task
- 3) if a subsystem contains unrelated objects, coherence is low
- 4) high cohesion is desirable

# Low Cohesion



# Example: High Cohesion



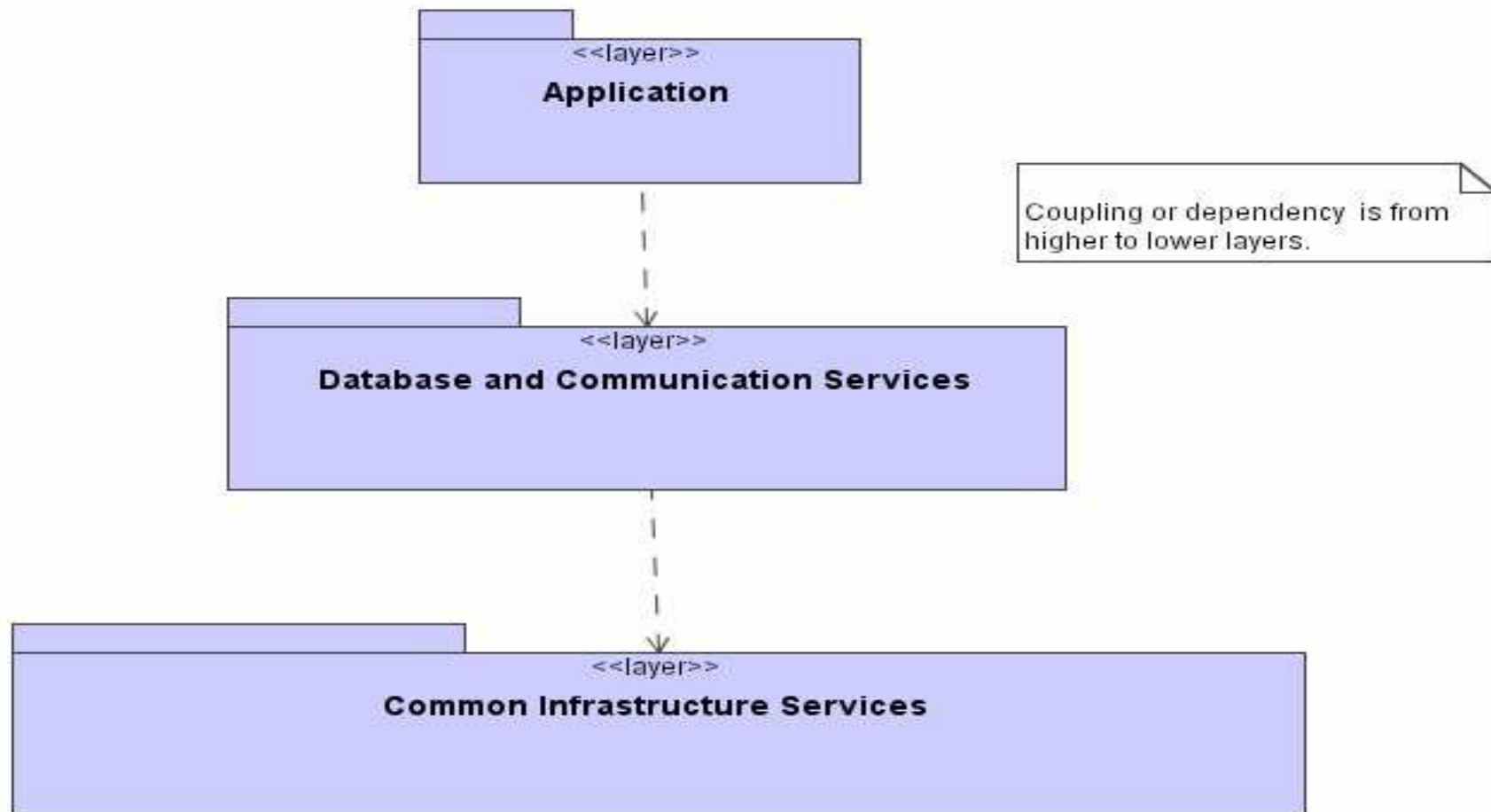


# Layering

---

- 1) A strategy for dividing system into subsystem.
- 2) Layering divides a system into a hierarchy of subsystems.
- 3) There are two common approaches to layering:
  - a) **responsibility driven**: layers have well-defined responsibilities, such that they fulfill specific roles in the overall scheme of things
  - b) **reuse driven**: layers are designed to allow for maximum reuse of system elements, by making higher level layers use services of lower level layers

# Example: Layered Architecture



# Packages

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary

# Packages

---

A general purpose mechanism for organizing modelling elements (e.g. classes) into groups.

It groups elements that are semantically close and that tend to change together.

Well structured packages are loosely coupled and very cohesive, with tightly controlled access to the package's content.



# Owned Elements

---

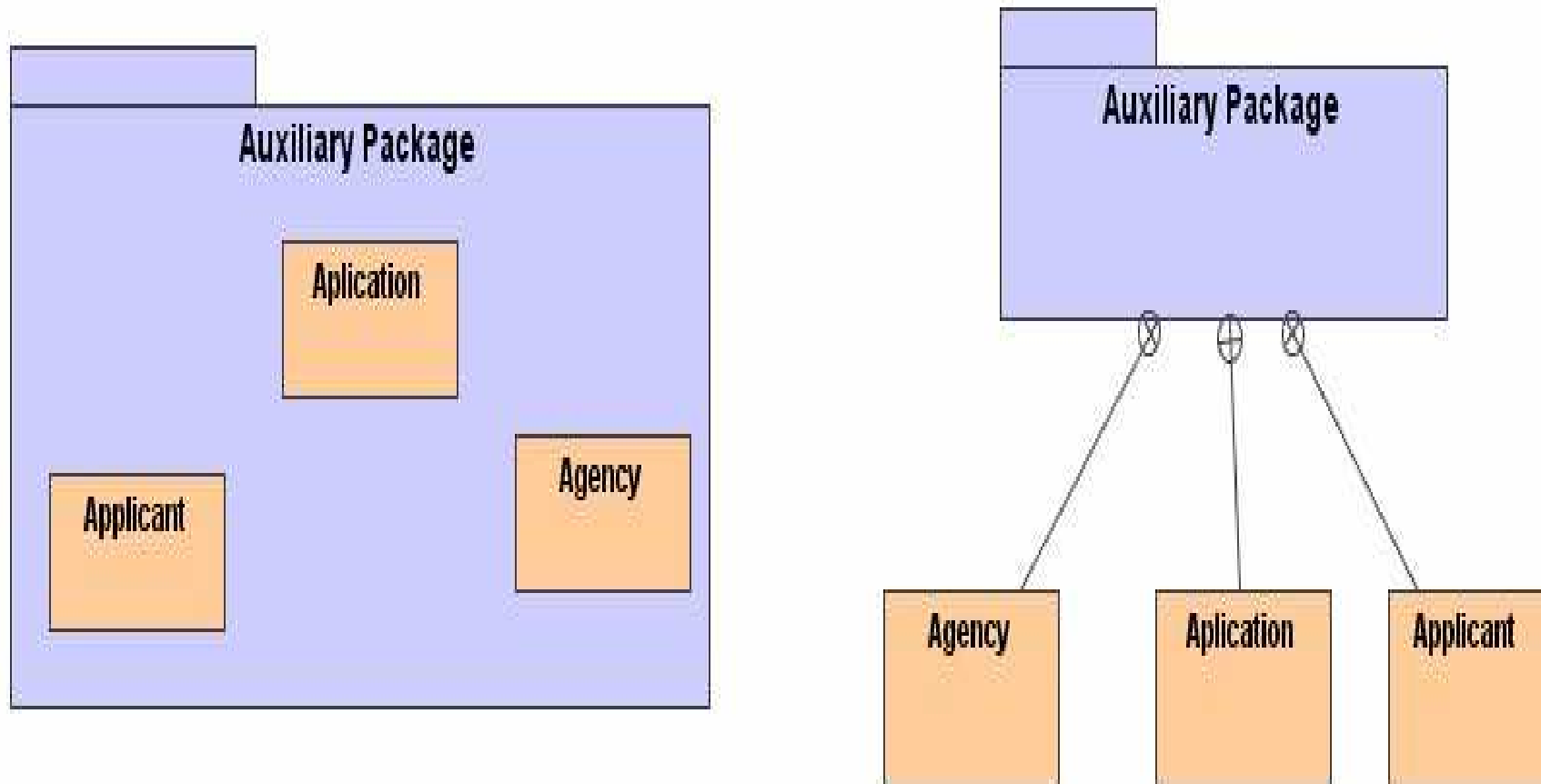
A package may own other elements for instance: classes, interfaces, components, nodes, collaborations, use cases, diagrams and other packages.

“Owning” is a composite relationship.

Elements are declared within the package and are destroyed when the package is destroyed.

A package forms a namespace, thus elements of the same kind must be named uniquely. For example you cannot have two classes in the same package with the same name.

# Example: Owned Elements



# Visibility

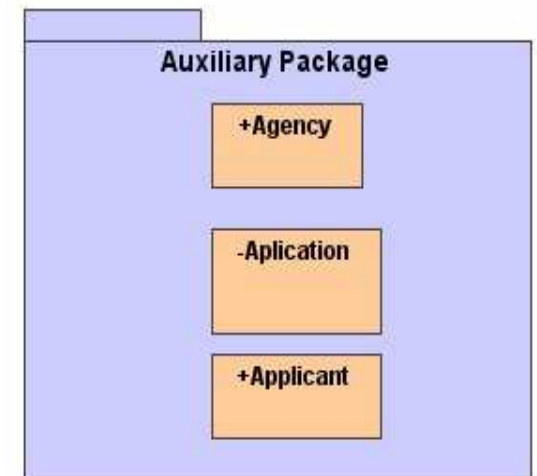
---

Visibility of owned elements can be controlled in a similar way as visibility for attributes and operations of classes.

Elements owned by the package is visible to contents of any package that imports the owning or enclosing package.

Protected elements can also be seen by children and private elements cannot be seen outside their owning package.

For example an element of any package that imports the "Auxiliary package" will have access to the "Agency" element but not "Application".





# Importing 1

---

Suppose *A* and *B* are two peer classes (*A* requires *B*), which are organized into separate packages.

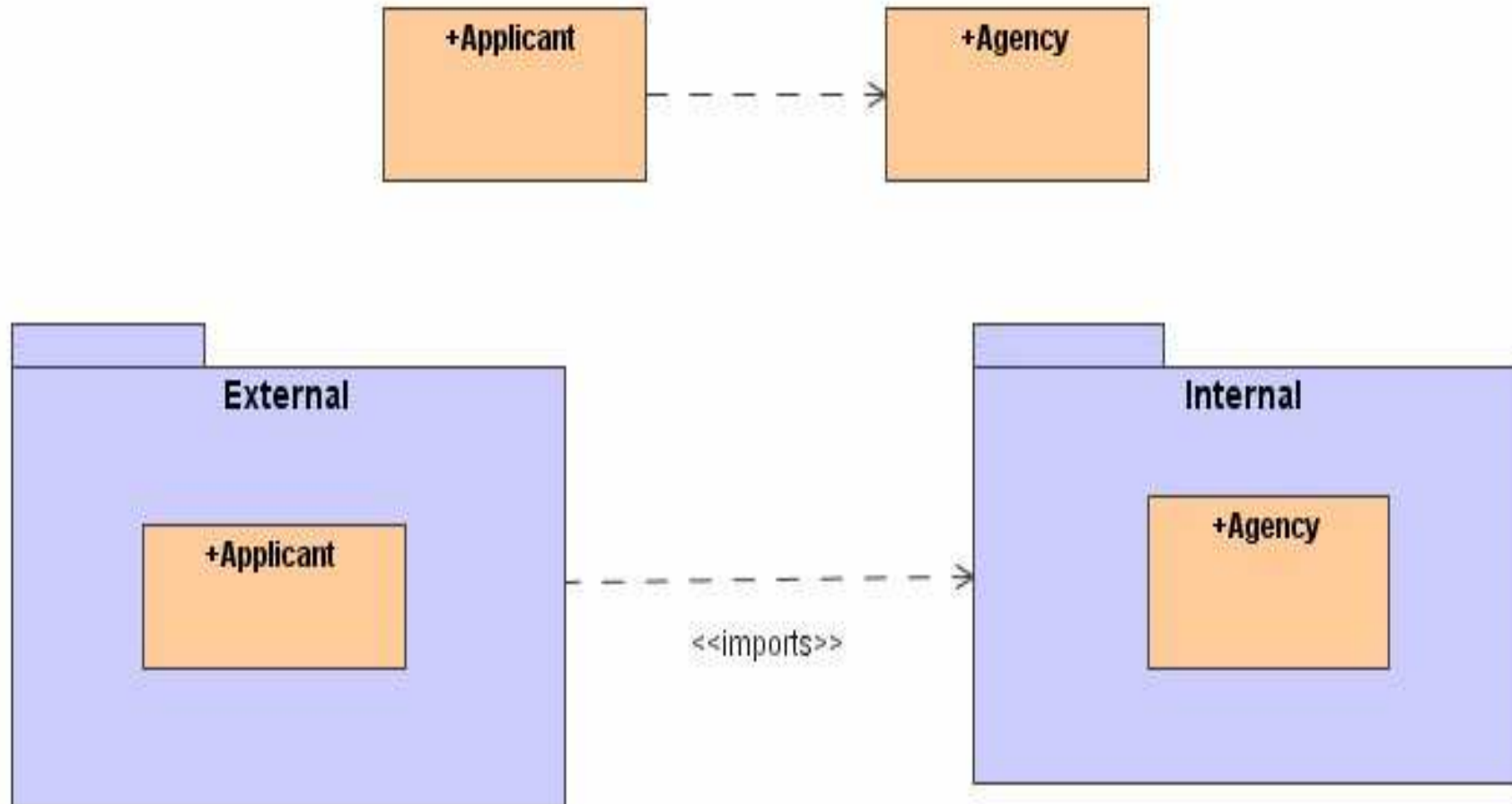
Also suppose that *A* and *B* are both declared as public in their respective packages.

Class *A* can only access class *B* when *A*'s package imports *B*'s Package.

Importing grants a one way permission for elements in one package to access elements in the imported package.

# Importing 2

---



# Exporting

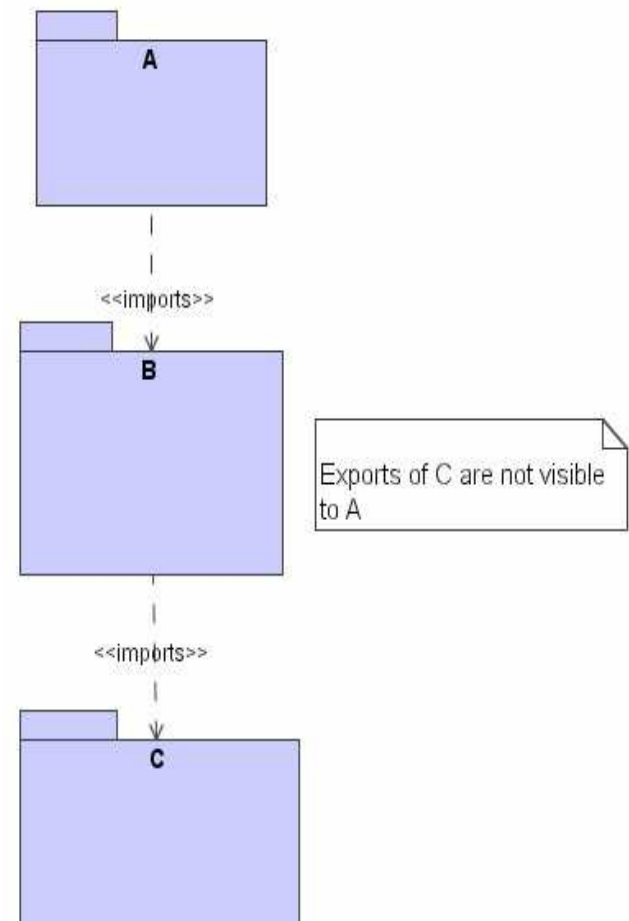
---

Exports are the contents of a package that are visible only to other packages that explicitly imports it.

For example *Agency* is an export of package *Internal*

Import and access dependencies are not transitive.

If an element is visible within a package, then it is visible to all packages nested within the package.



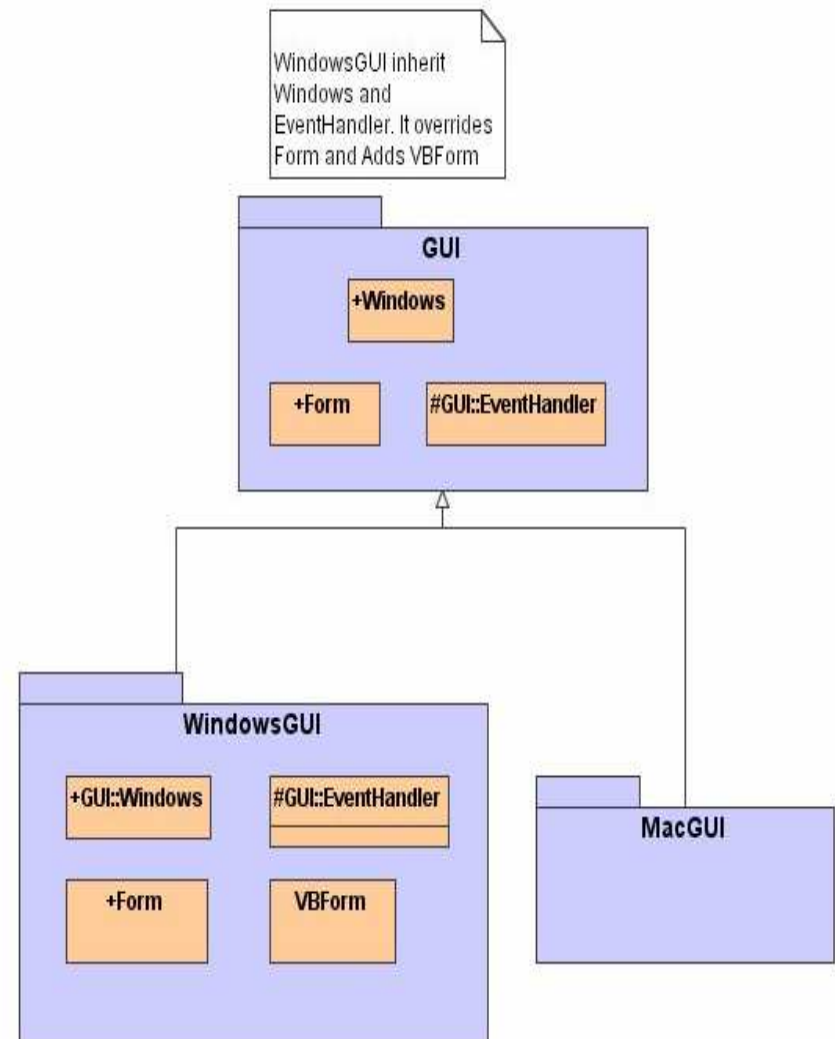
# Generalization

Generalization is the second kind of relationship that can exist between packages.

Generalization is used to specify a family of packages and is similar to generalization between classes.

Packages involved in generalization relationships follow the same substitutability rule as classes.

*WindowsGUI* may be used wherever *GUI* is used.



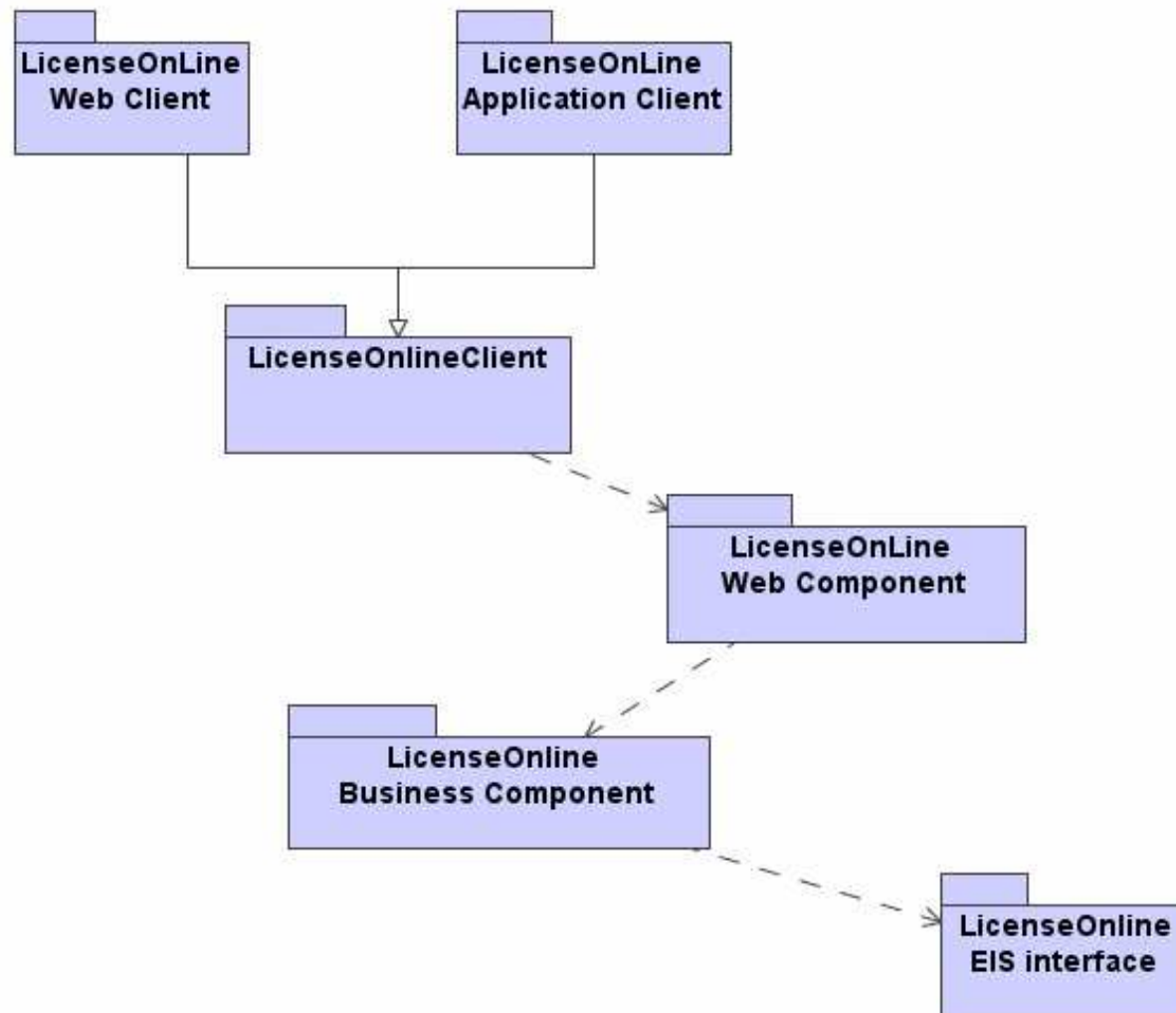
# Package Stereotypes

UML provides five stereotypes that apply to packages:

- 1) **Façade**: a package that is only a view on some other packages
- 2) **Framework**: a package that consists mainly of patterns
- 3) **Stub**: specifies a package that serves as a proxy for the public contents of another package
- 4) **Subsystem**: a package representing an independent part of the entire system being modelled
- 5) **System** : specifies a package representing the entire system being modelled

# Case Study: Packages

---



# Collaboration Diagrams

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary



# Collaboration

---

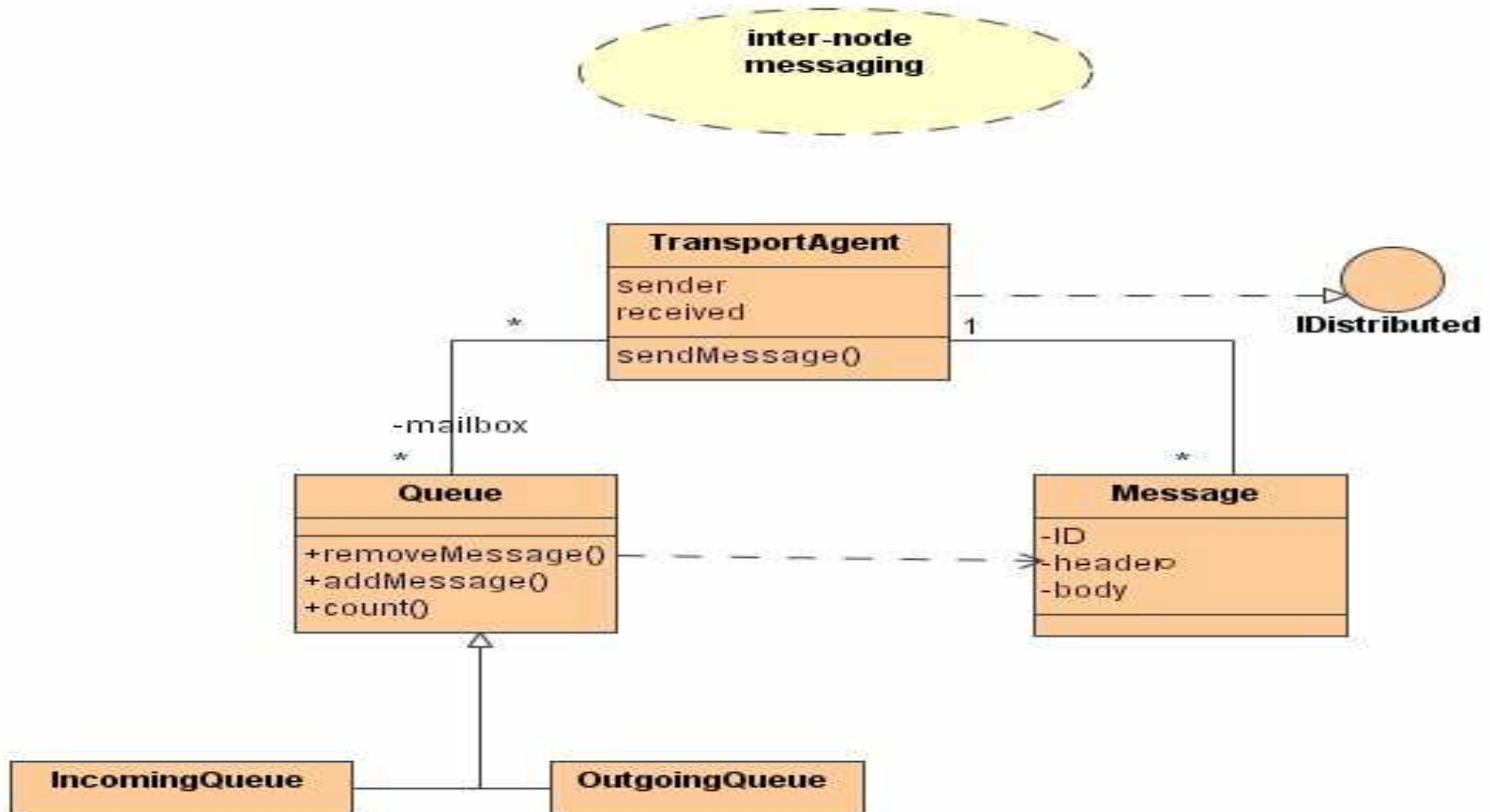
## Definition

A **collaboration** is a society of classes, interfaces, and other elements that work together to deliver or provide some cooperative behaviour that is bigger than the sum of all its parts.

Collaboration in the context of software architecture allows you to name a **conceptual** chunk that encompasses both static and dynamic aspects.

It specifies the realization of a use case and operations by a set of classifiers and associations playing specific roles used in a specific way.

# Example: Collaboration



*Inter-node Messaging Collaboration and its details*

# Collaboration Names

---

Every collaboration has a name that distinguishes it from other collaborations.

Collaboration names are nouns or short noun phrases, typically first letter of the noun is capitalized.

Example: "*Inter-node Messaging*" or "*Application Submission*"

# Collaboration – Structural

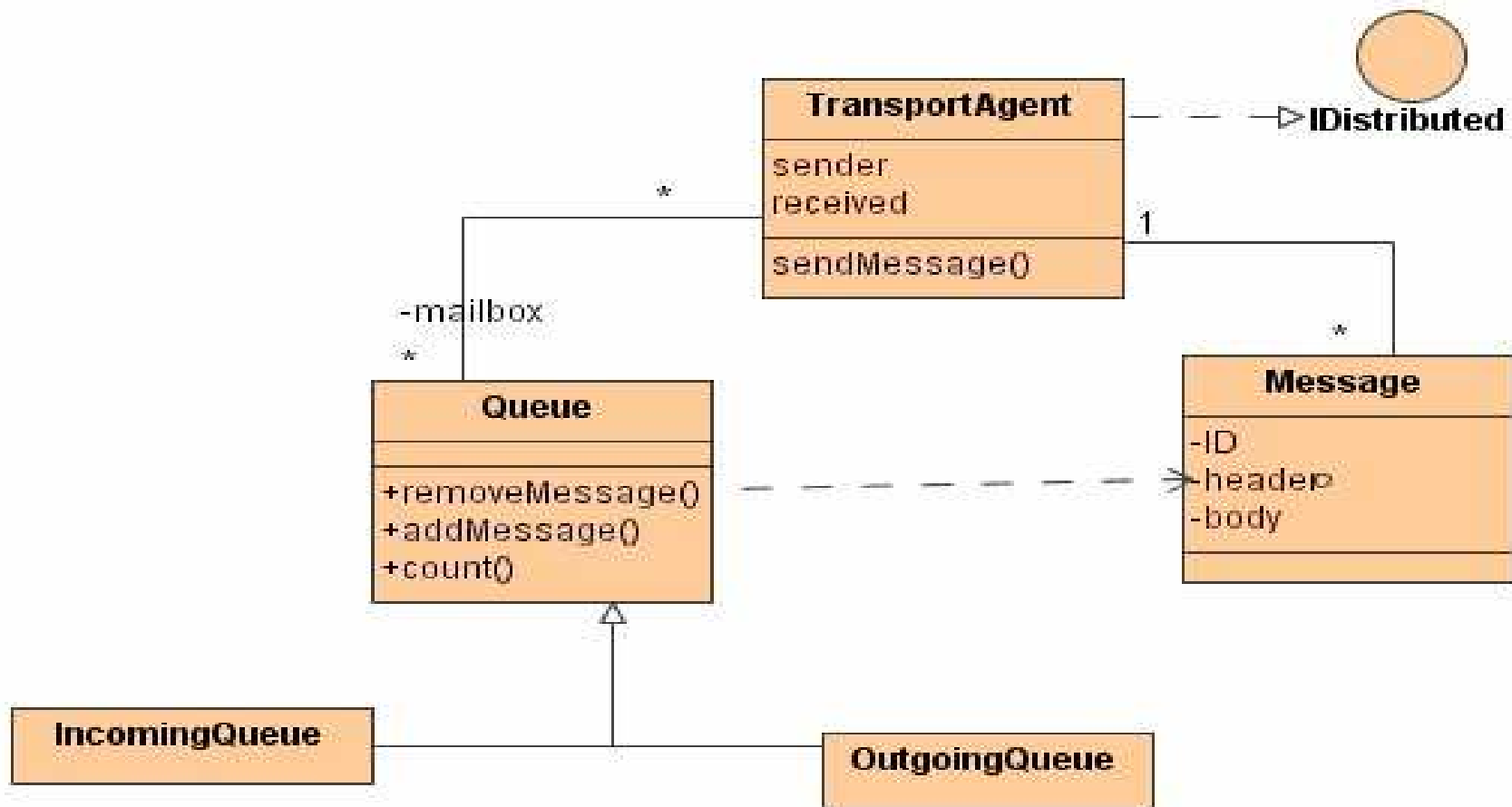
Structural aspects of collaborations specifies the classes, interfaces, and other elements that work together to carry out a named collaboration.

It includes a combination of classifiers, such as classes, interfaces, components and nodes.

Classifiers (Classes, components, nodes etc.) may be organized using all the usual UML relationships including association, generalization, and dependencies.

Unlike packages or subsystems, a collaboration does not own any of its structural elements , it only references the classes, interfaces, components etc. declared elsewhere.

# Example: Collaboration 1



*Structural View of Collaboration*

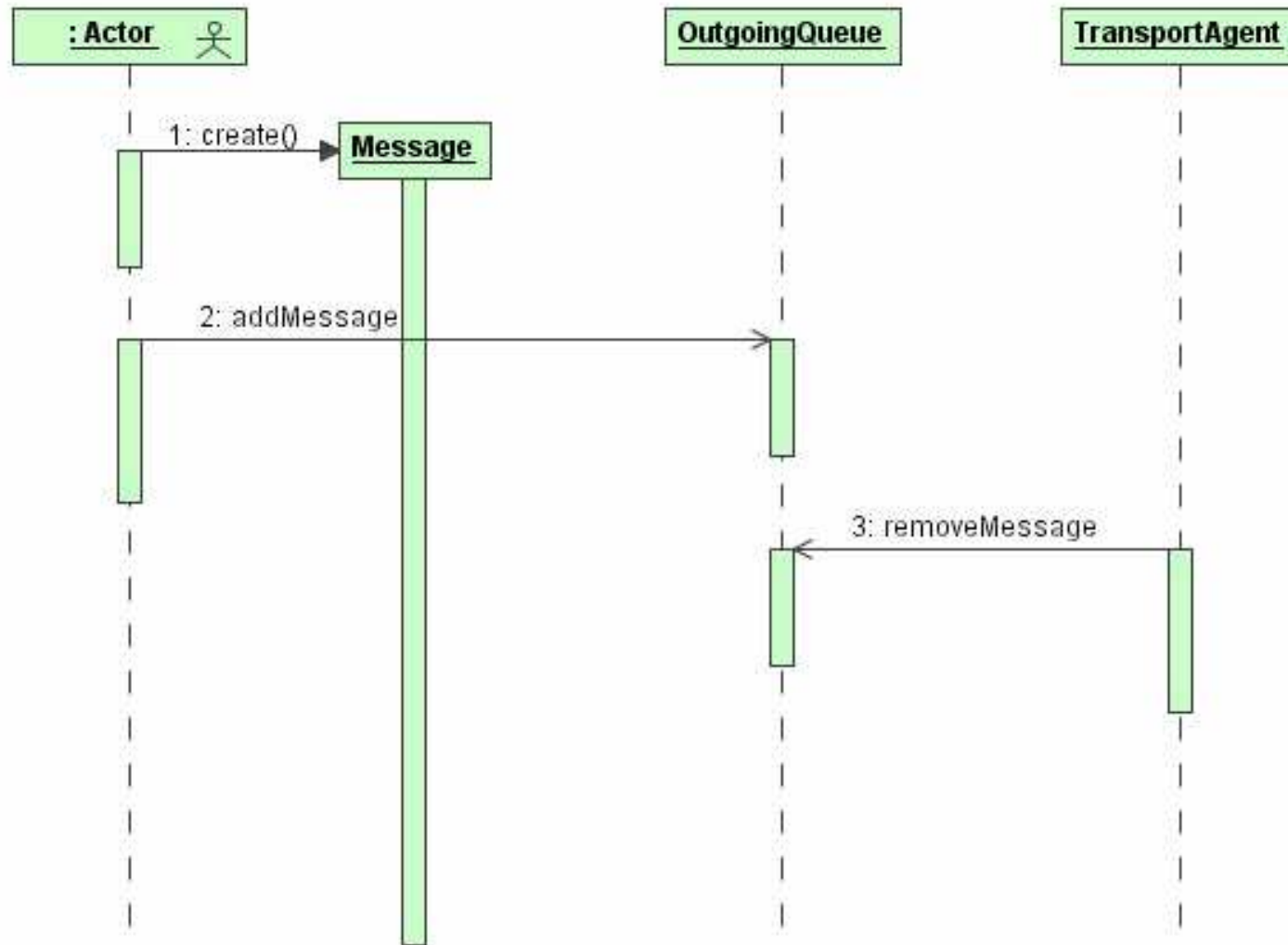
# Collaboration – Behavioural

The behavioural aspect of a collaboration is rendered using interaction diagram.

An interaction diagram specifies an interaction that represents a behaviour composed of a set of messages that are exchanged among a set of objects to accomplish a specific purpose.

An interaction context is provided by its enclosing collaboration which establishes the classes, interfaces, components, nodes and other structural elements whose instances participate in that interaction.

# Example: Collaboration 2



# Organizing Collaborations

Collaborations are essential in system architectures.

A well structured OO system is composed of modestly sized regular set of collaborations.

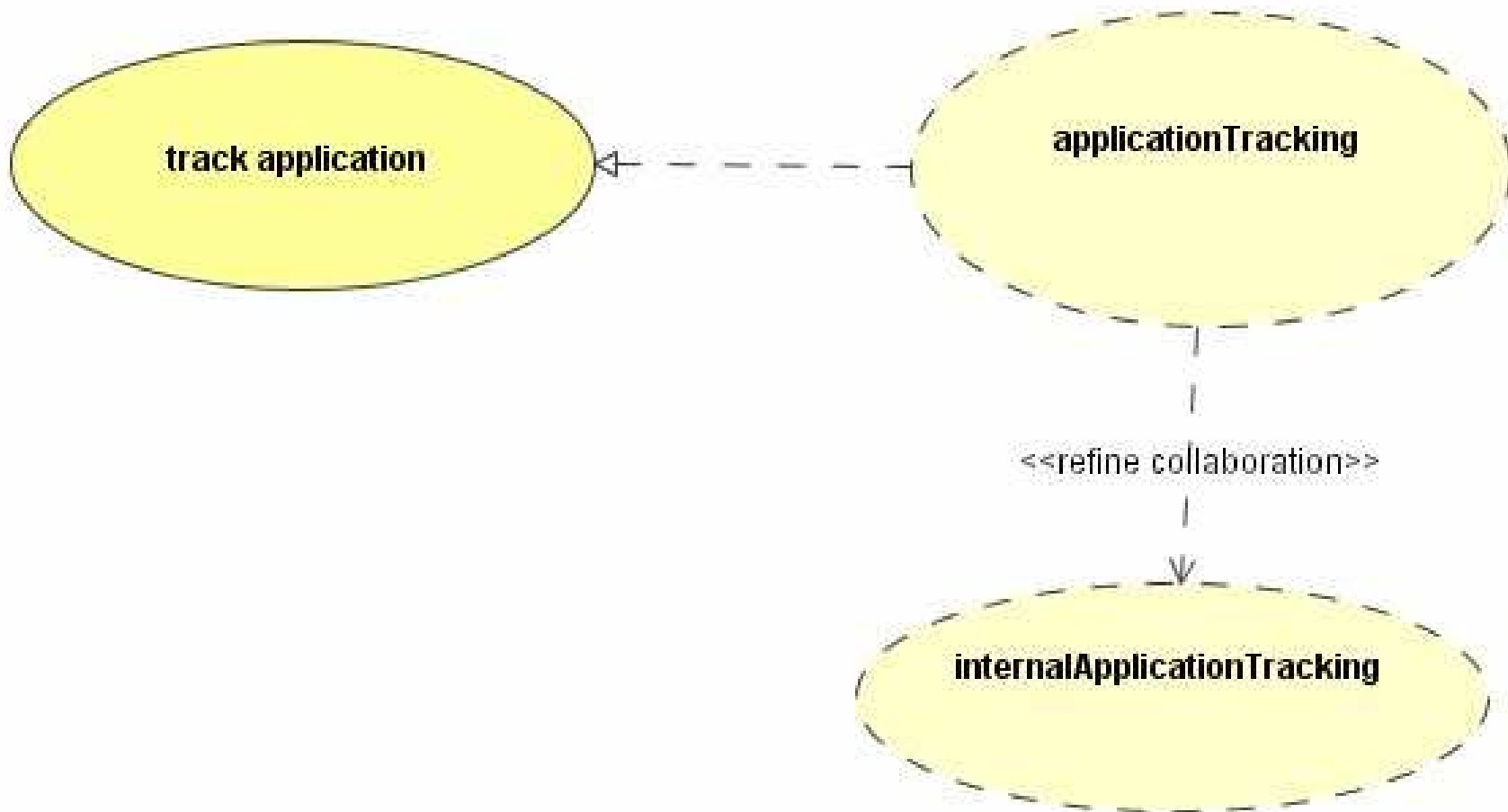
There are two sets of relationships in collaborations:

- a) relationship between collaboration and what it realizes (use cases or operations)
- b) relationship between collaborations

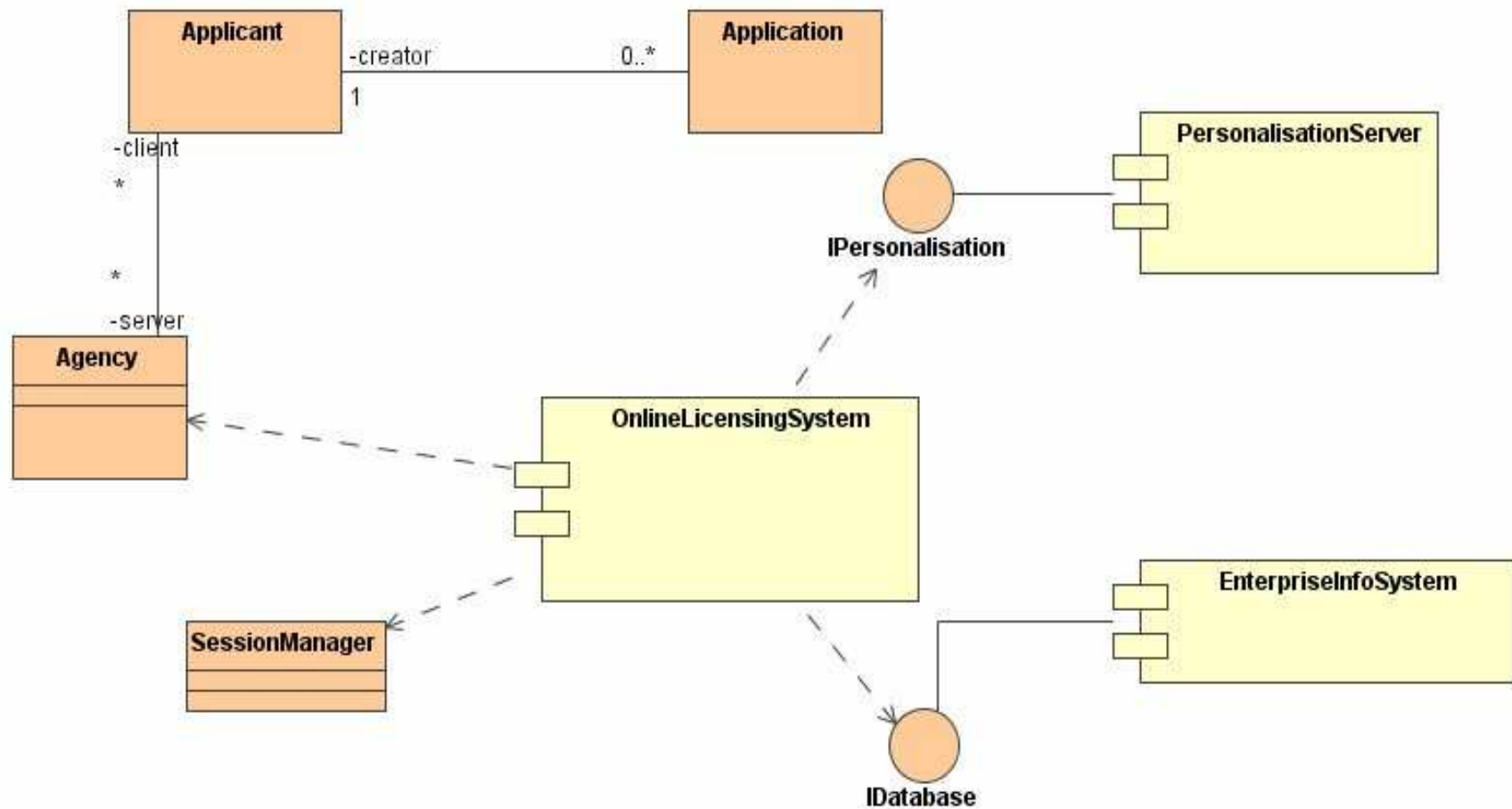
These relationships correspond to realization and refinement respectively.



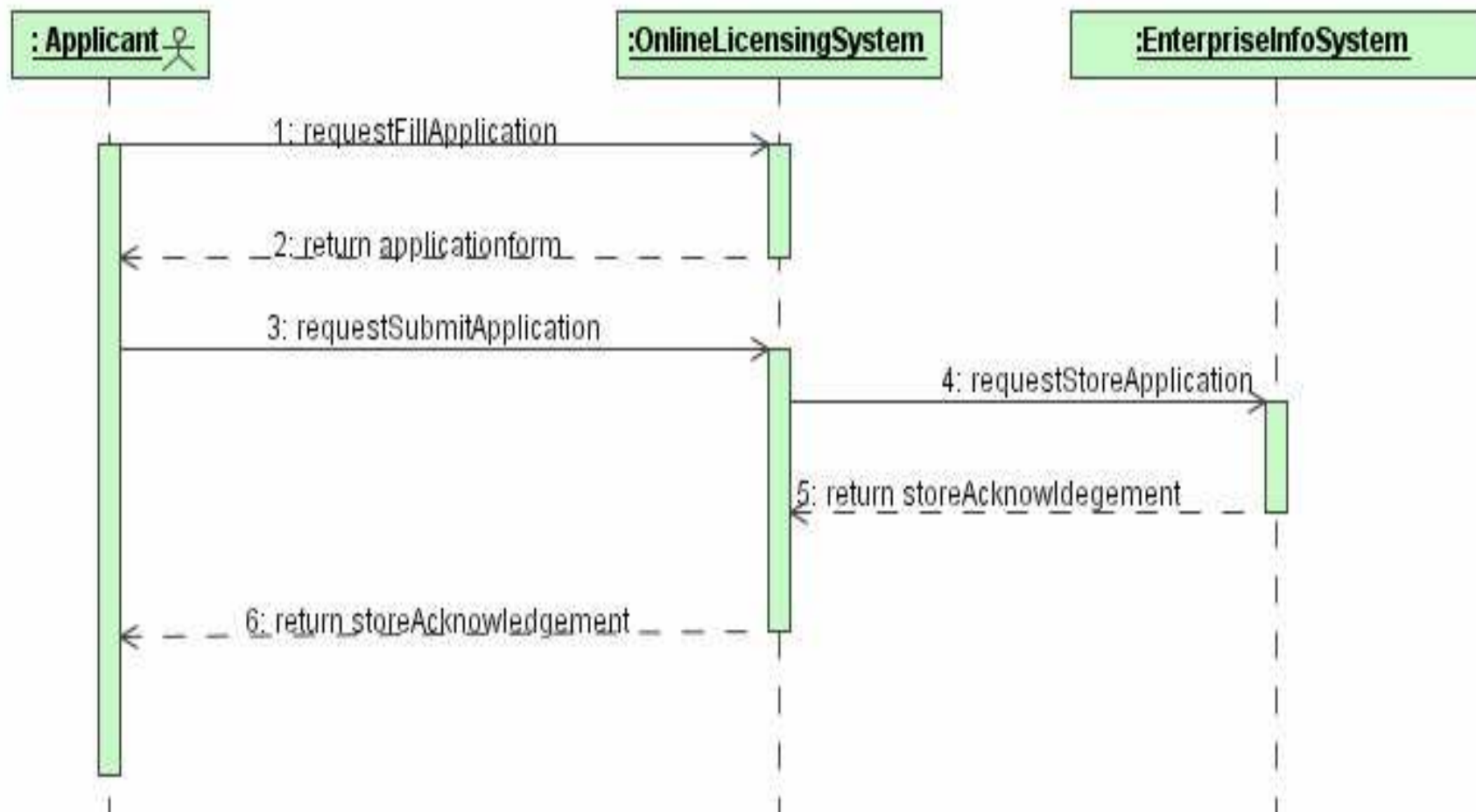
# Organizing Collaborations



# Case Study: Structural



# Case Study: Behavioural



*Behavioural View of a Collaboration which implements a “submit application online” use case*

# Collaboration Diagram

---

## Definition

A collaboration diagram shows the interactions organized around the structure of a model, using either classifiers (e.g. classes) and associations or instances (e.g. objects) and links.

It presents a collaboration, containing a set of roles to be played by instances as well as their required relationships given in a particular context.

It also presents an interaction which defines a set of messages (stimuli) specifying the interaction between instances playing a certain role within a collaboration to achieve a desired result.

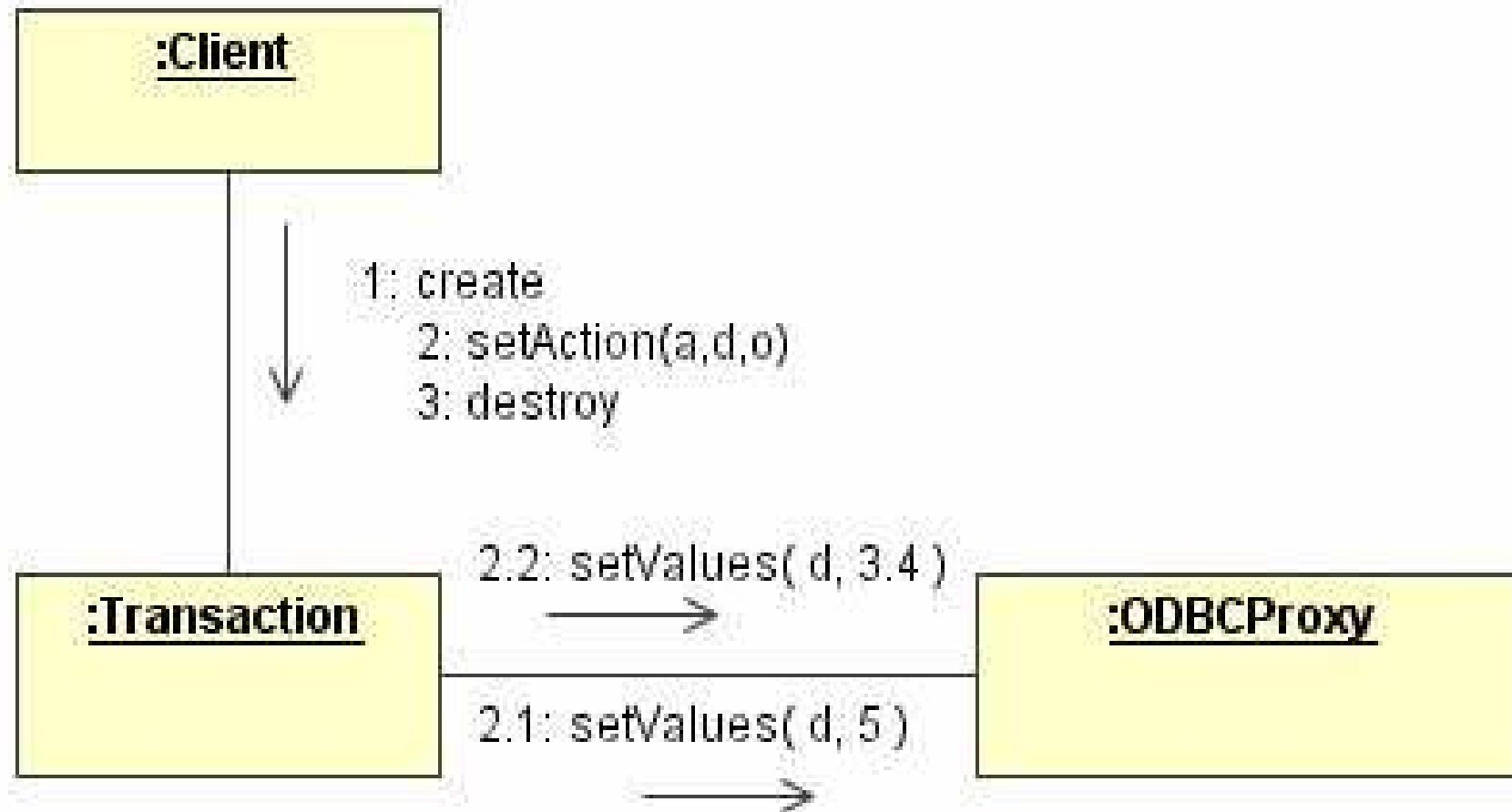
# Notation 1

---

- 1) A collaboration diagram shows a graph of either instances linked to each other or classifiers and associations.
- 2) Navigability is shown using arrow heads on the lines representing links.
- 3) An arrow next to a line indicates a stimuli or message flowing in the given direction.
- 4) The order of interaction is given with a number starting from *1*.

# Notation 2

---



# Notation 3

---

A collaboration diagram without any interaction shows the “context” in which an interaction can occur.

A collection of standard constraints may be used to show whether an instance or a link is created or destroyed during the execution:

- {new} – instances and links created during execution
- {destroyed} – instances and links destroyed during execution
- {transient} - instances and links created and destroyed during the execution

# Collaboration Diagrams Types

Collaboration diagrams may be presented at two levels:

- 1) instance level – instances and stimuli
- 2) specification levels – roles

Instance level (or context):

- defined in terms of a static structure of instances and their relationships and possibly the stimuli

Specification level:

- shows a collaboration



# Instance Diagram

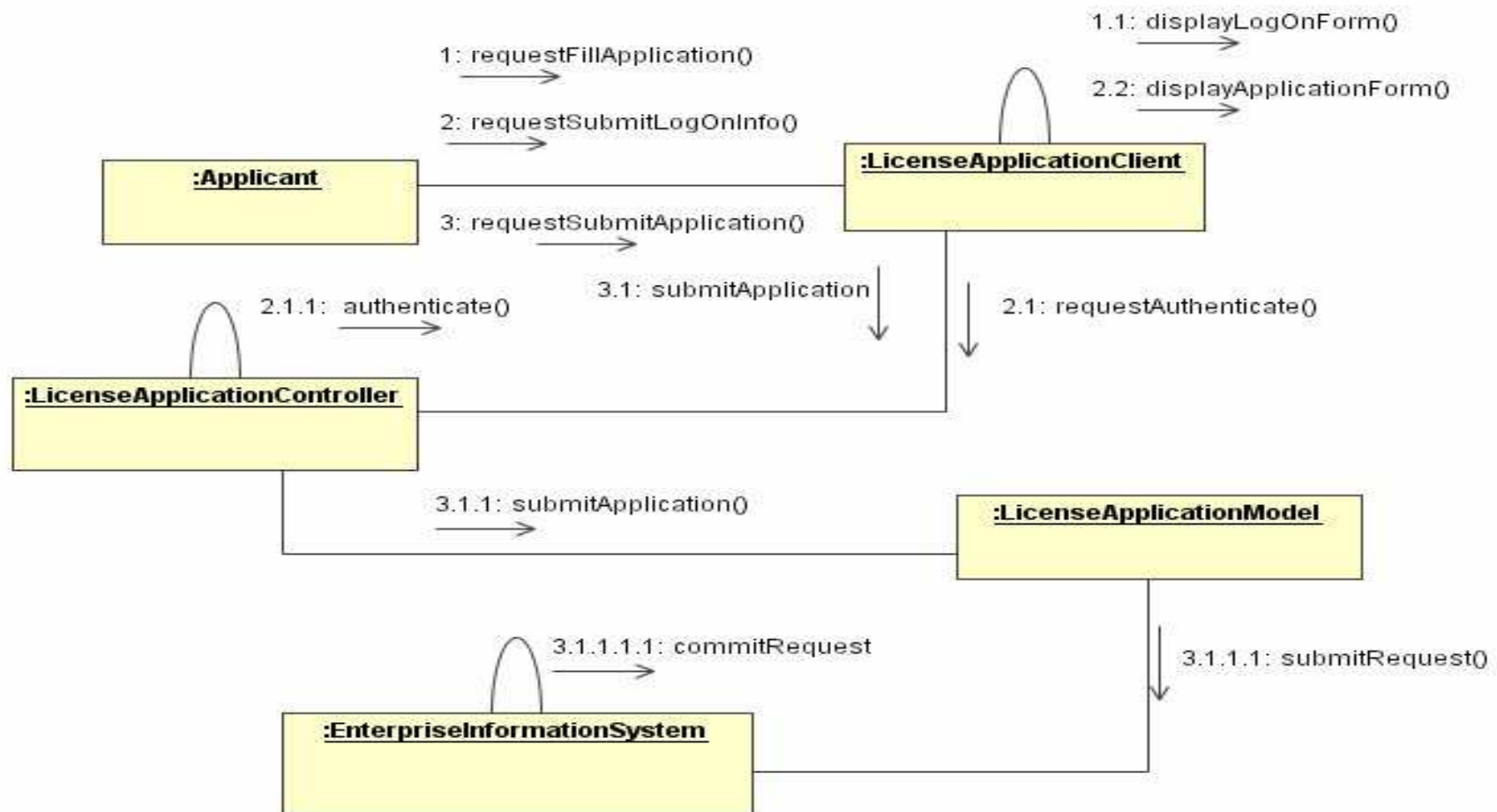
---

It shows a CollaborationInstanceSet – a collection of object boxes and lines mapping to instances and links, respectively.

It may include arrows attached to lines that corresponds to stimuli communicated over links.

It also shows the instances relevant to the realization of an Operation.

# Example: Instance Level



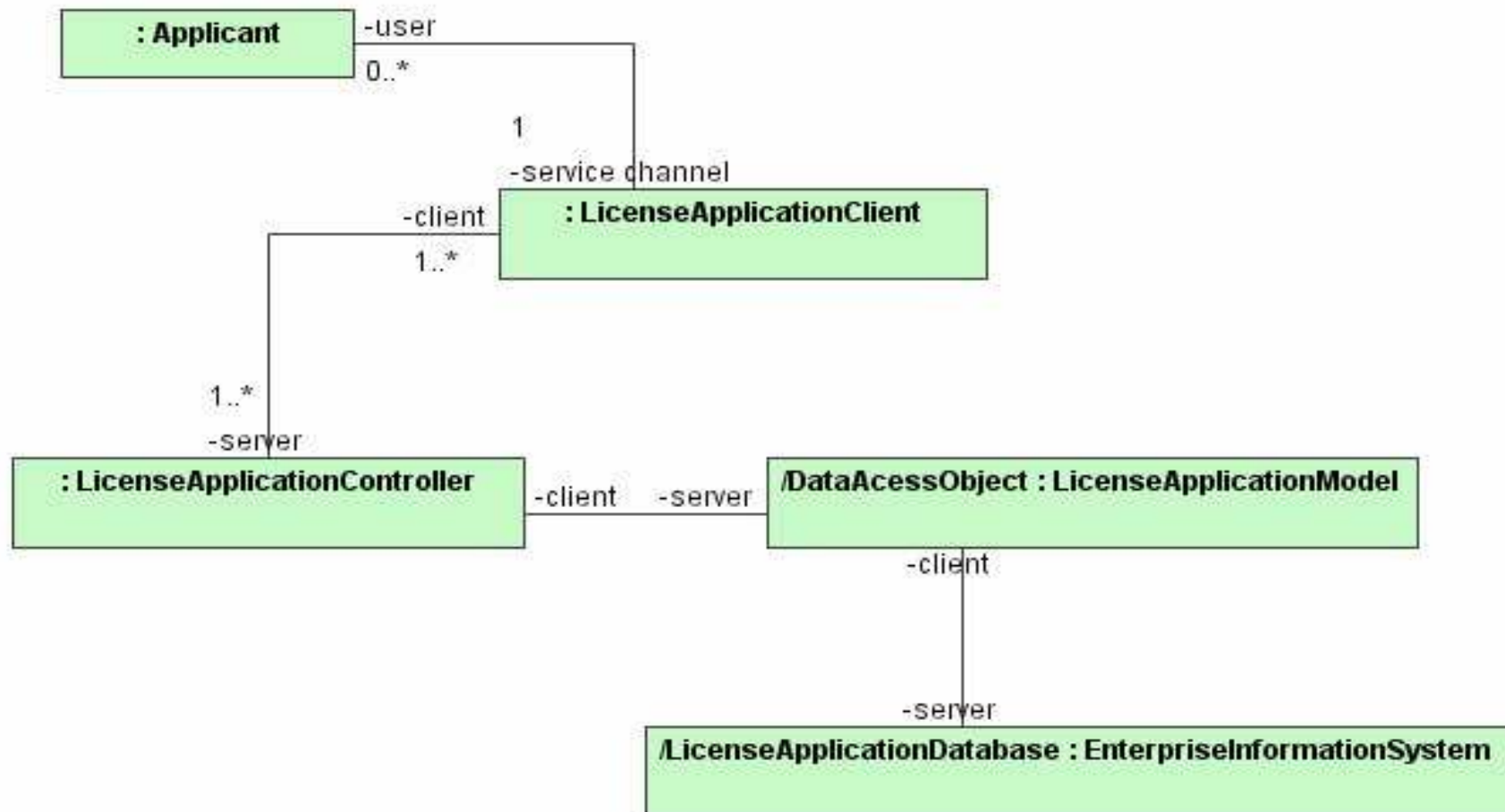
# Specification Diagram

It shows a collaboration.

It provides the roles defined within a collaboration.

The arrows attached to lines map into messages.

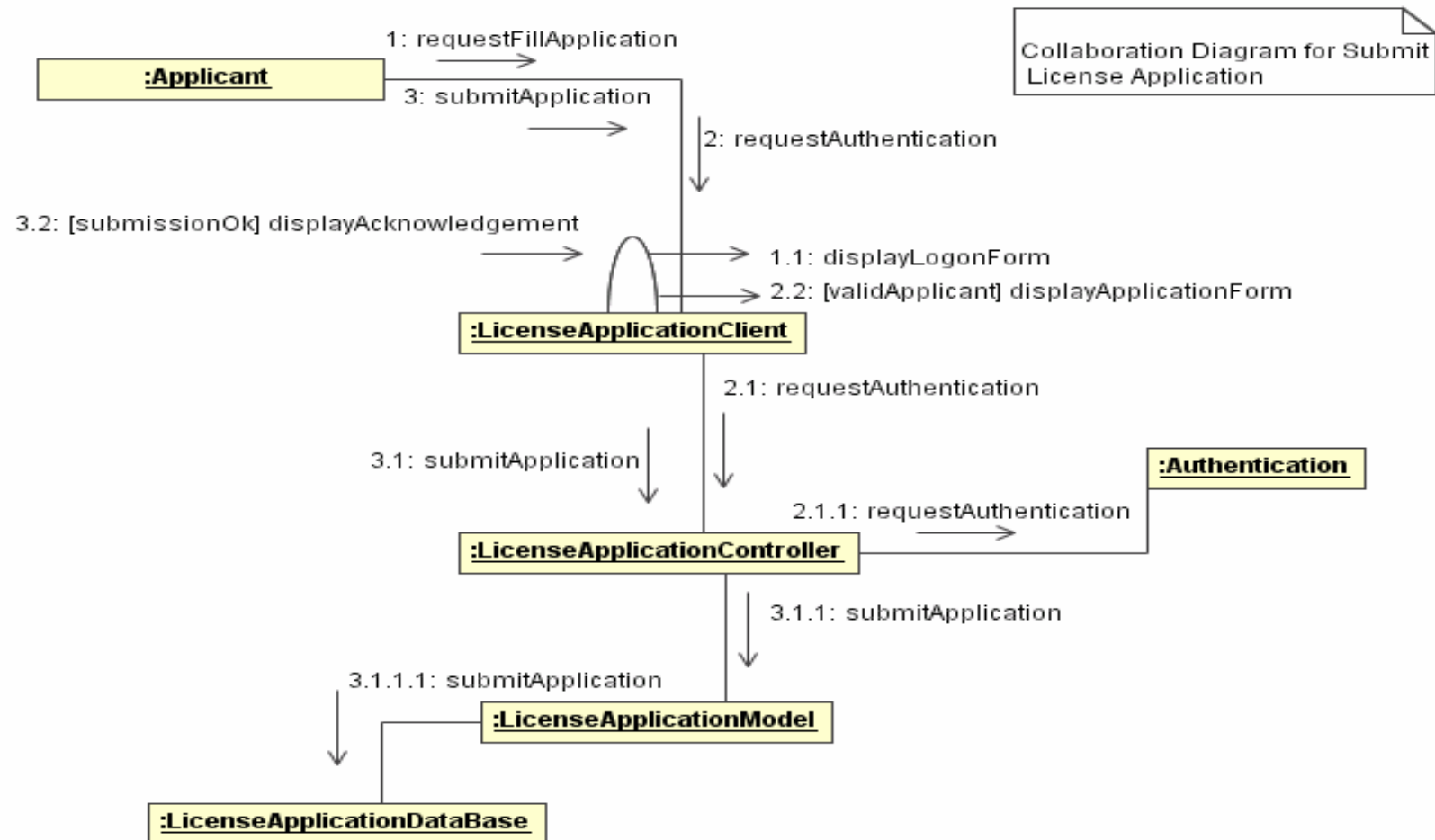
# Example: Specification Level



# Collaboration Roles Syntax

Syntax	Explanation
:C	Un-named instance originating from the Classifier C
/R	Un-named instance playing the role R
/R:C	Un-named Instance originating from C, playing role R
O/R	Instance named O playing role R
O:C	Instance named O originating from the Classifier C
O/R:C	An instance named O originating from the Classifier C, playing role R
O	An instance named O

# Example: Case Study



# Component Diagrams

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary



# Component

---

## Definition

A **component** is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

It encapsulates the implementation of classifiers residing in it.

A single element may reside in multiple components.

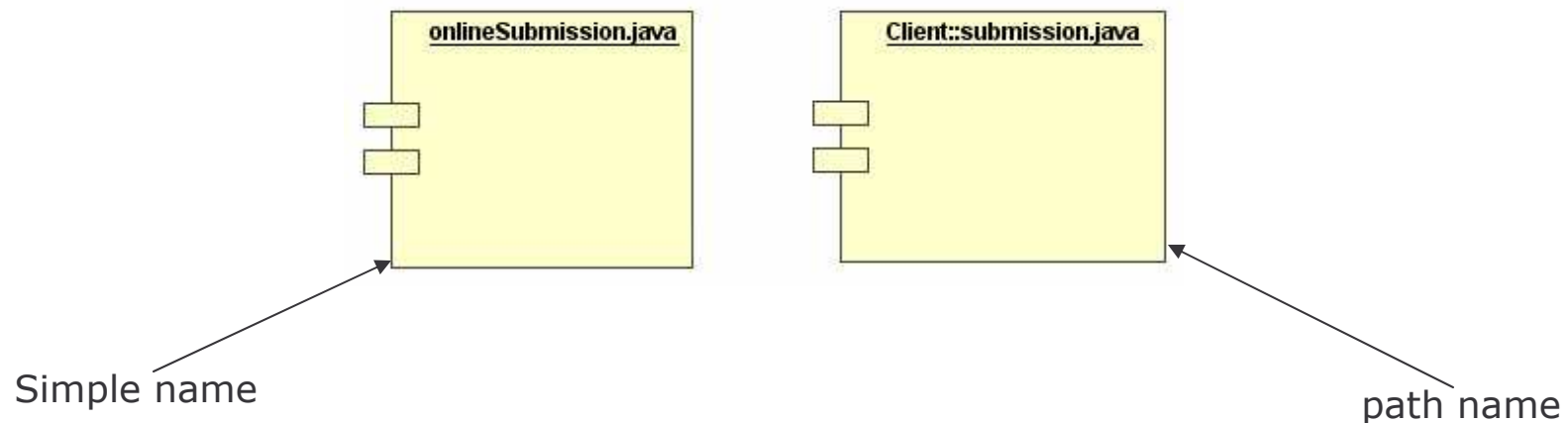
A component does not have its own features, but a mere container for its elements.

Components are replaceable or substitutable.

# Component Names

Every component must have a name that distinguishes it from other components.

Name is a textual string which may be written as a simple name or path name.



# Component and Classes 1

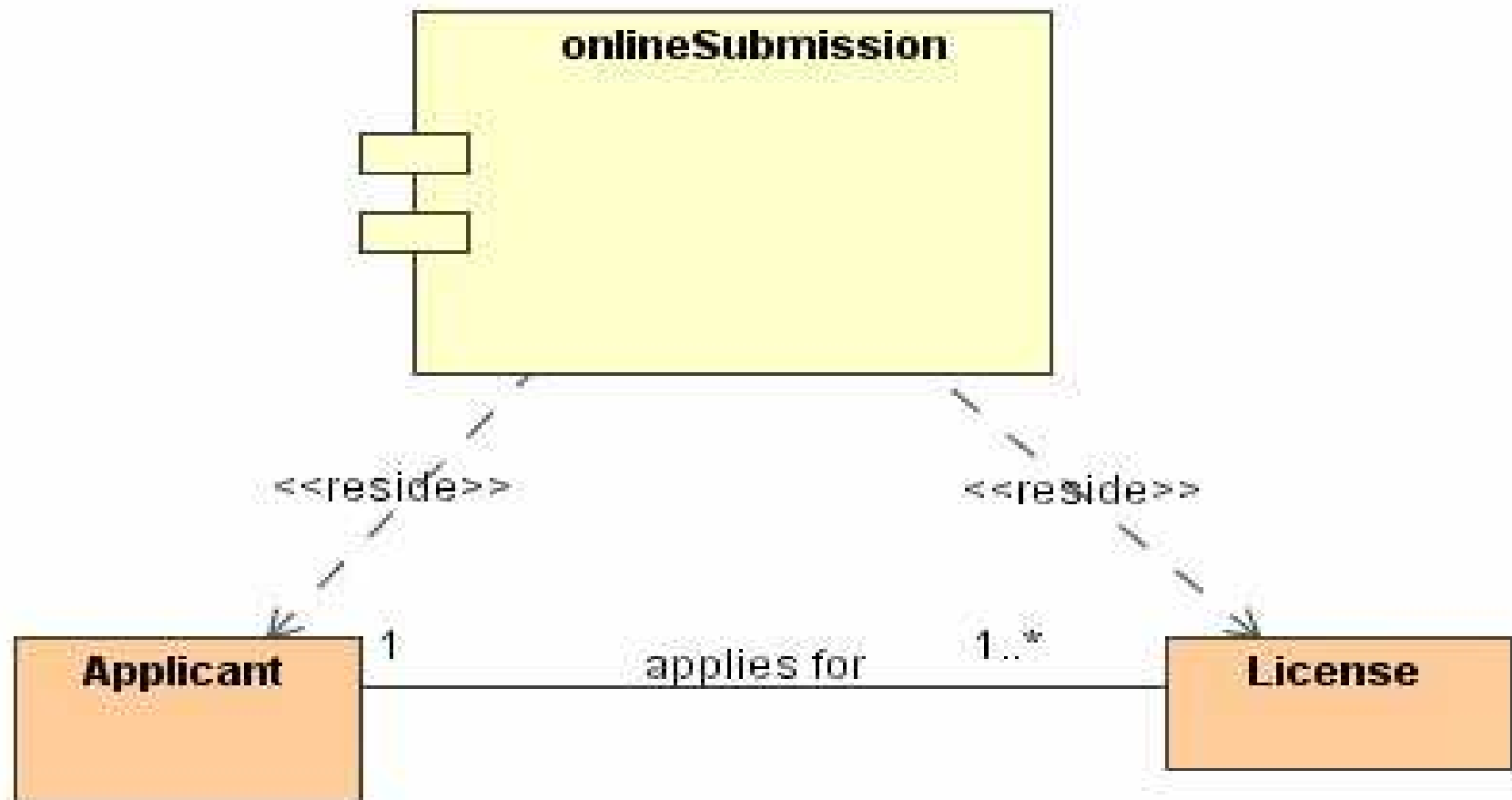
## similarities

- 1) both may realize a set of interfaces
- 2) both may participate in dependencies, generalizations and associations
- 3) both may be nested
- 4) both may have instances
- 5) both may participate in an interaction

## differences

- 1) classes represent logical abstraction while components represent physical things
- 2) components represent the physical packaging of logical components and are at a different level of abstraction
- 3) classes may have attributes and operations whereas components only have operations reachable only through their interfaces

# Component and Classes 2



# Components and Interfaces

## Definition

An **interface** is a collection of operations that are used to specify a service of a class or components.

Using component technologies (COM+, CORBA, EJB), physical implementation may be decomposed by specifying interfaces that represent the major seams of the system.

Components that realize these “seams” are created or provided in implementation.

Interfaces allow for the deployment of systems whose services are somewhat location independent and replaceable.

# Component Notation 1

Relationship between a component and its interface may be shown in two ways:

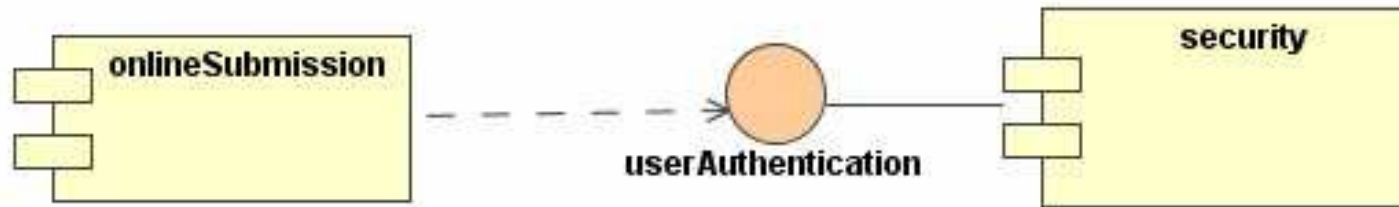
## 1) style 1

- a) interface in elided iconic form
- b) component that realize the interface is connected to the interface using an elided realization relationship

## 2) style 2

- a) interface is presented in an enlarged form, possibly revealing its operations
- b) realizing component is connected to it using a full realization relationship

# Component Notation 2



Style 1: elided iconic or short hand



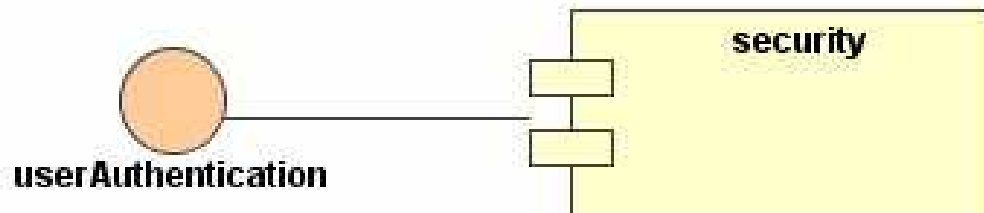
Style 2: long hand

# Export Interfaces

## Definition

An interface that a component realizes is called an export interface, meaning an interface that the component provides as a service to other components.

Components may export more than one interfaces.



The userAuthentication Interface is an export interface of the security component.



# Import Interface

---

## Definition

An interface that a component uses is called the import interface, meaning the interface that the component conforms to and builds on.

Components may import more than one interface.

They may also import and export interfaces at the same time.



The lpersonalisation Interface is an Import Interface for the OnlineSubmission Component

# Component Replaceability

The key intent of any component based OS facility is to permit the assembly of system from binary replaceable parts.

A system can be:

- a) created out of components
- b) evolved by adding new components and replacing old ones without rebuilding the system

Interfaces allow easy reconfiguration of component based systems.

# Types of Components

There are three basic types of components.

1) **Deployment:**

component necessary and sufficient to form an executable system such as DLLs and EXEs

2) **Work product component:**

residue of development process consisting of things like source code files and data files from which deployment components are created

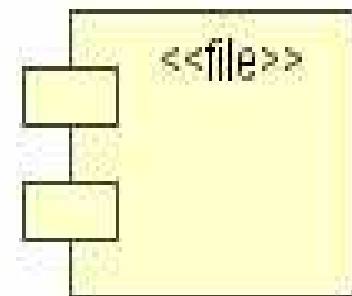
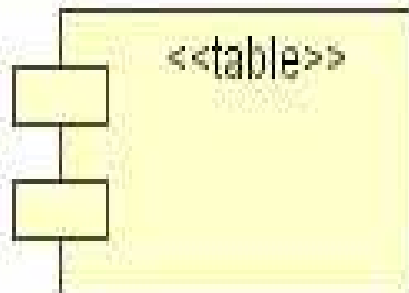
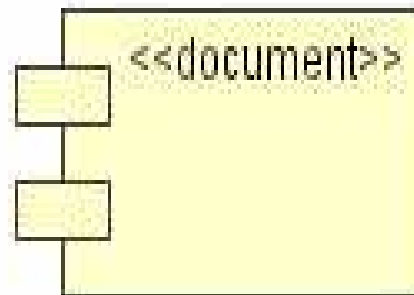
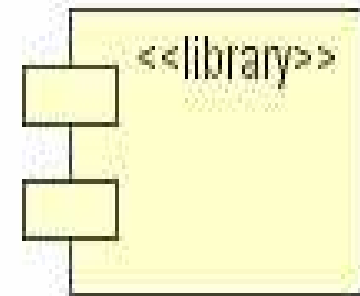
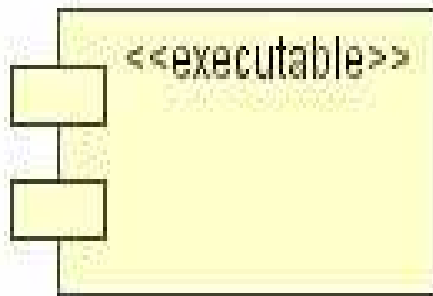
3) **Execution component:**

created as a consequence of an executing system

# Component Stereotypes 1

- 1) **executable**: specifies that a component may be executable on a node
- 2) **library**: specifies a static or dynamic object library
- 3) **table**: specifies a component that represent a database table
- 4) **file**: specifies a component that represents a document containing source code or data
- 5) **document**: specifies a document that represent a document

# Component Stereotypes 2



# Component Diagram

---

## Definition

A **component diagram** shows the dependencies among software components including the classifiers that specify them and the artifacts that implements them.

It models the physical implementation of the software specified by the logical requirements of the class diagram.

It indicates the relationship between the classes that specify the requirements for the components and the artifacts that implements them.

# Component Diagram Notation 1

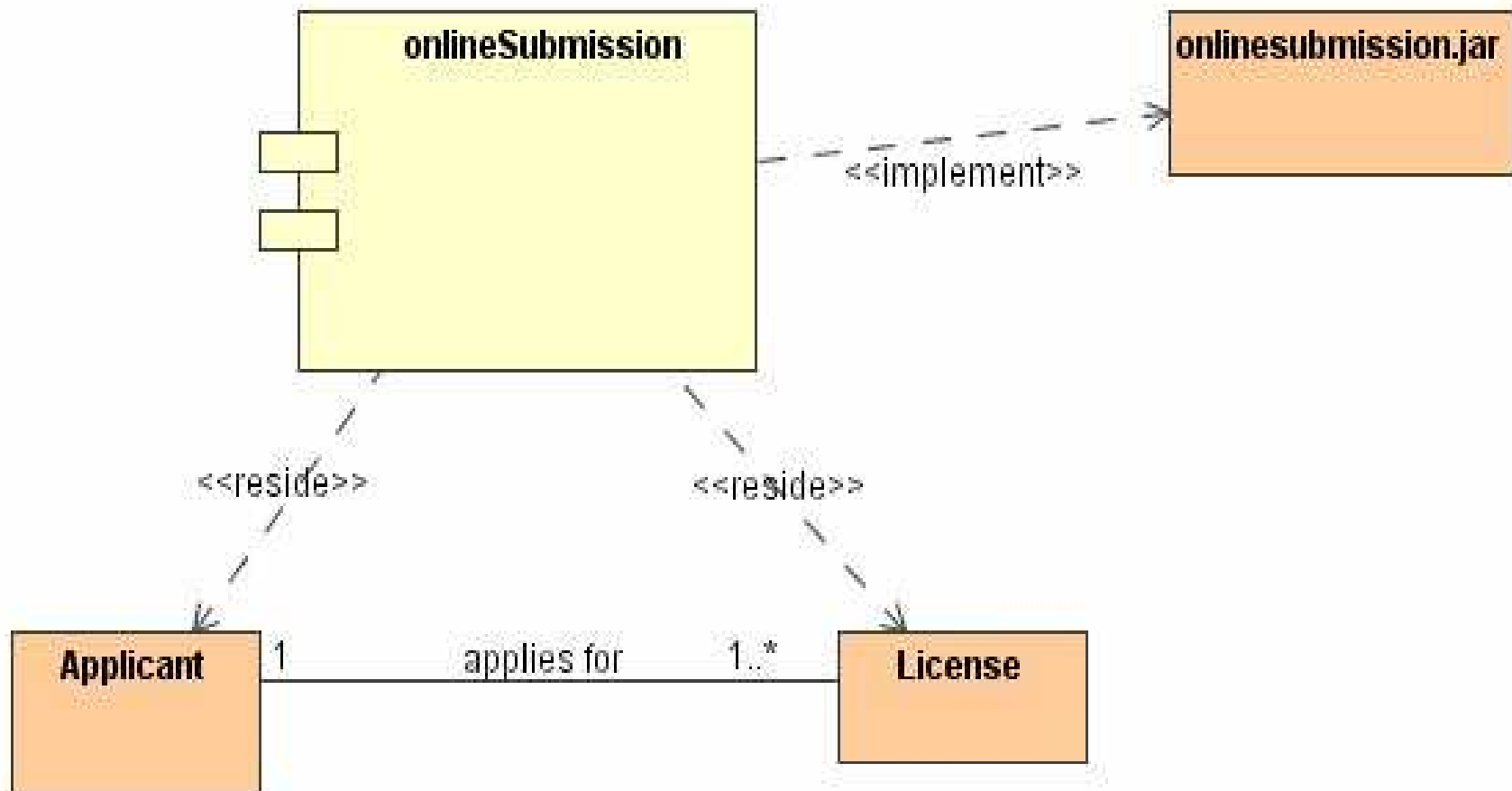
A graph of components, classifiers and artifacts connected by dependency relationships.

Components may also be connected using physical containment representing composition.

Classifiers that specify the components can be connected to them by physical containment or by a `<<reside>>` relationship.

Artifacts that specify components can be connected to them by an `<<implement>>` relationship.

# Component Diagram Notation 2

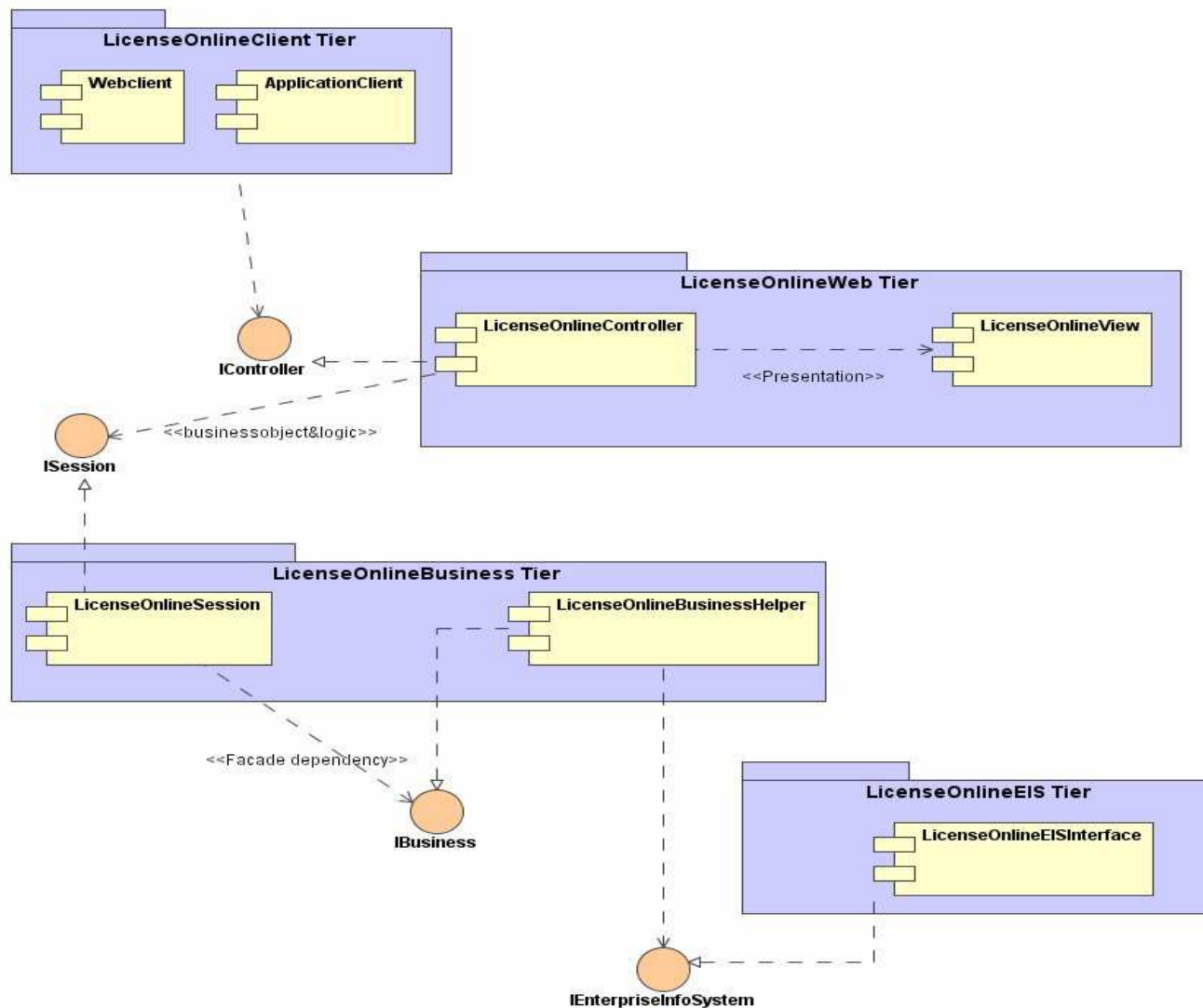




# Component Diagram Content

- 1) specifying classes
- 2) implementing artifacts
- 3) components
- 4) interfaces
- 5) dependencies, generalization, association and realization relationships

# Example: Component Diagram



# Architectural Patterns

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary

# Templates

---

## Definition

A **template** is a parameterized element with one or more unbound parameters defining a family of elements.

Templates are used to generalize the structure and behaviour of a society of elements.

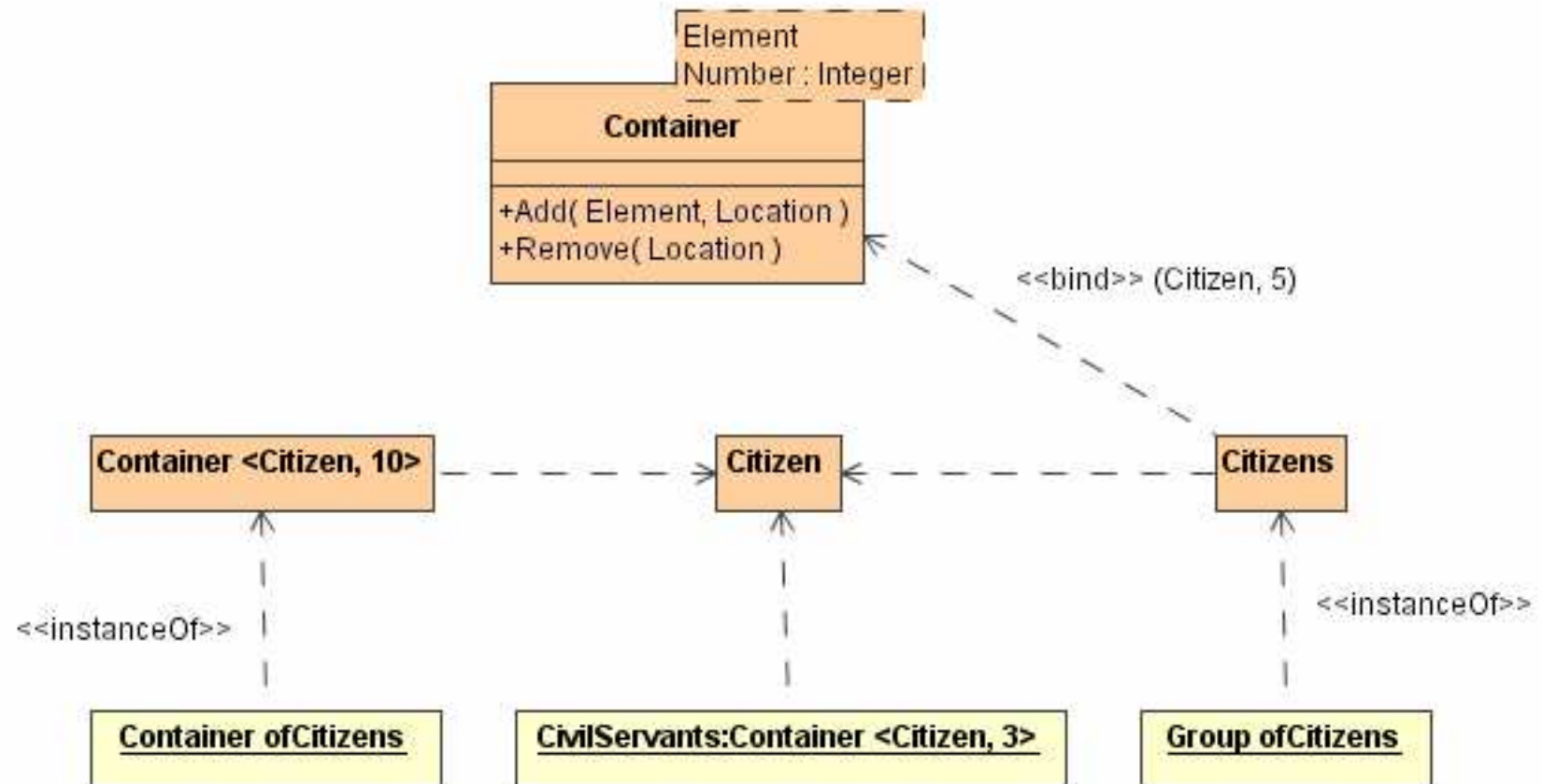
A template is not directly usable, because it has unbound parameter and must be bound to actual values before it is used through a binding relationship.

# Template Notation

---

- 1) a template class is depicted with a small dashed rectangle superimposed on the upper right hand corner of the rectangle for the class
- 2) the formal parameters are comma separated while the actual parameters are specified through the <<bind>> dependency relations
- 3) formal parameters may also be listed one per line including the parameter's name, implementation type and an optional value
- 4) a missing type for a formal parameter indicates a class type

# Example: Template



# Patterns – What are they?

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way the you can use this solution a million times over, without doing it the same way twice.”

– Christopher Alexander (Patterns in buildings and towns)



# Patterns Definition

---

## Definition

A generalized solution to a problem in a given context where each pattern has a description of the problem, solution context in which it applies, and heuristics including use advantages, disadvantages and trade-offs.

Patterns identify, document, and classify best practices in OOD.

They generalize the use and application of a society of elements.

Solution is described using static structure, identifying participating elements and their relationships, and a dynamic behaviour, identifying how elements collaborate and interact.

# Notation and Description

---

## Notation:

a pattern is depicted as a template or parameterized collaboration, where formal parameters are roles used in the collaboration

The **description** of a pattern uses four elements:

- a) pattern name
- b) problem
- c) solution
- d) consequences

# Pattern Elements

---

- 1) **name**: a handle which describes the design problem, its solution, and consequences in a word or two
- 2) **problem**: describes when to apply the pattern and explains the problem and its context
- 3) **solution**: elements that must make up the design, their relationships, responsibilities and collaborations.
- 4) **consequences**: associated trade-offs in using the pattern.

# Example: Pattern 1

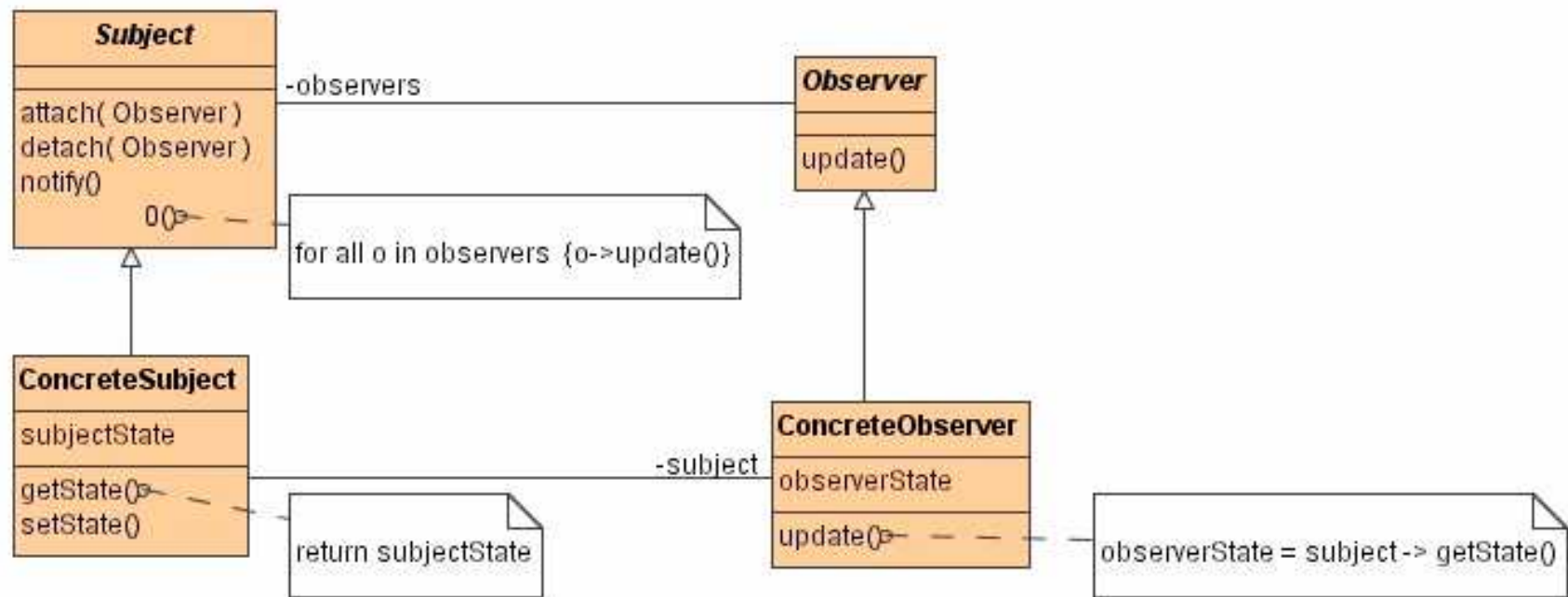
1) **Name:** Observer

2) **Problem:**

- a) When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
- b) When a change to one object requires changing others, and you don't know how many objects need to be changed
- c) When an object should be able to notify other object without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

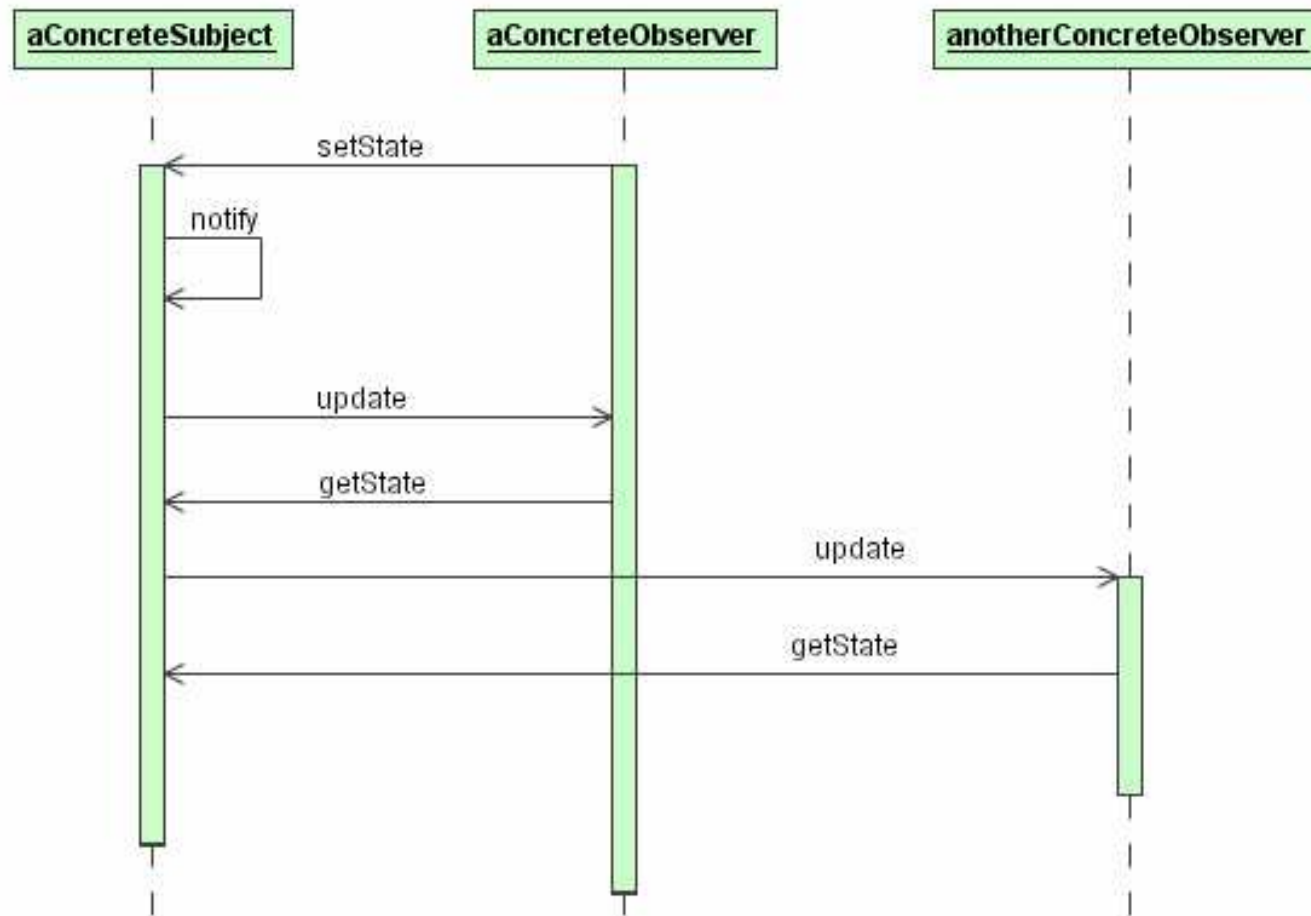
# Example: Pattern 2

## 3) Solution – Structural:



# Example: Pattern 3

## 3) Solution - Behavioural:



# Example: Pattern 4

---

## 4) Consequences:

- a) abstract coupling between Subject and Observer
- b) support for broadcast communication
- c) unexpected updates

# Types of Patterns

---

## Creational Patterns:

apply to instantiation of objects and are concerned with decoupling the type of objects from the process of constructing that object.

## Structural and Architectural patterns:

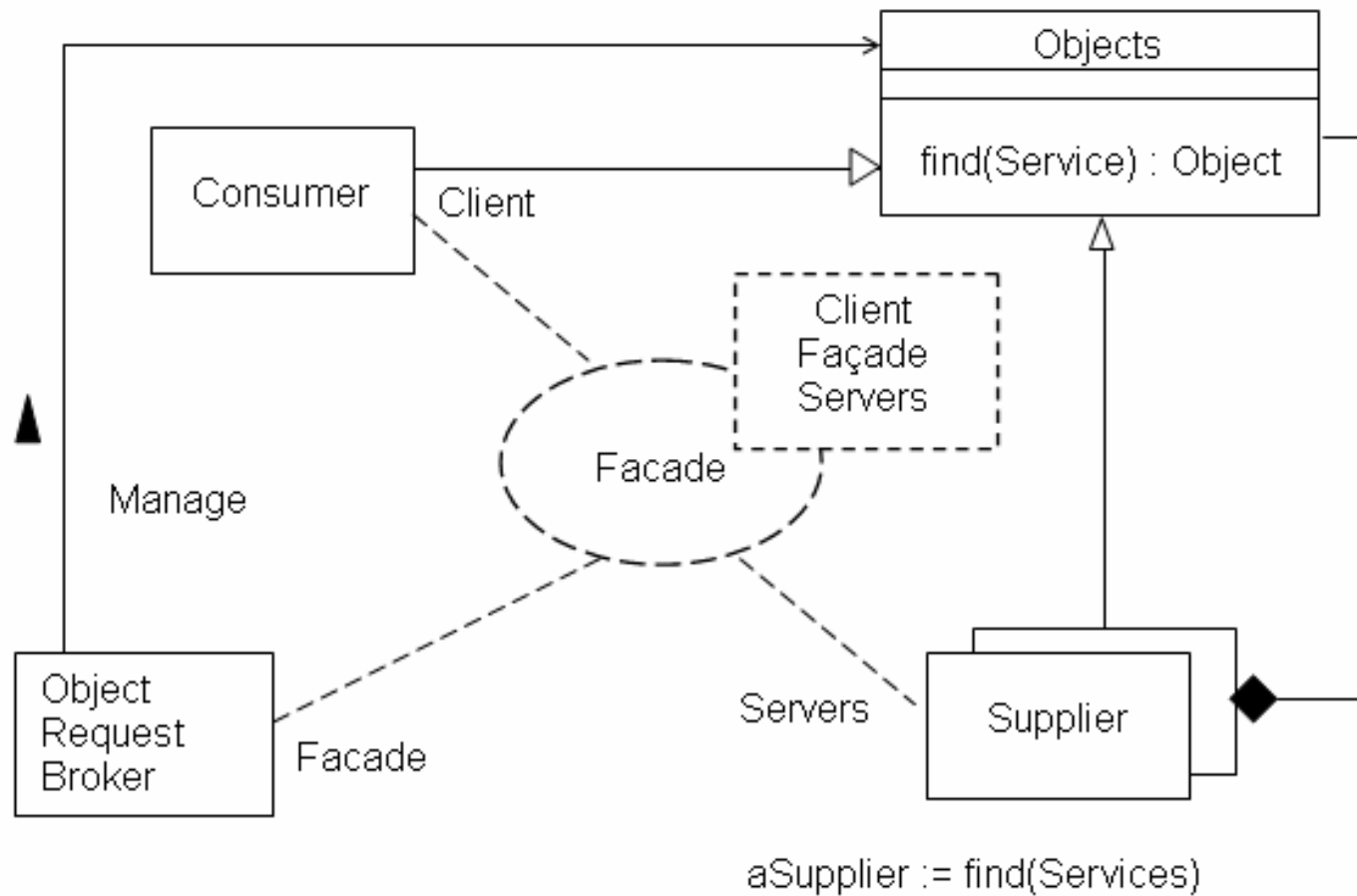
apply during the organization of a system and concerned with larger structures composed from smaller structures.

## Behavioural Patterns:

assigning responsibilities among a collection of objects.



# Example: Façade Pattern



# Framework 1

---

## Definition

A **Framework** is a reusable software architecture that provides the generic structure and behaviour for a family of software applications, along with a context that specifies their collaboration and use.

A framework is also a collection of patterns defined as template collaborations with supporting elements.

It is a skeletal solution in which specific element must be plugged in order to establish an actual solution.

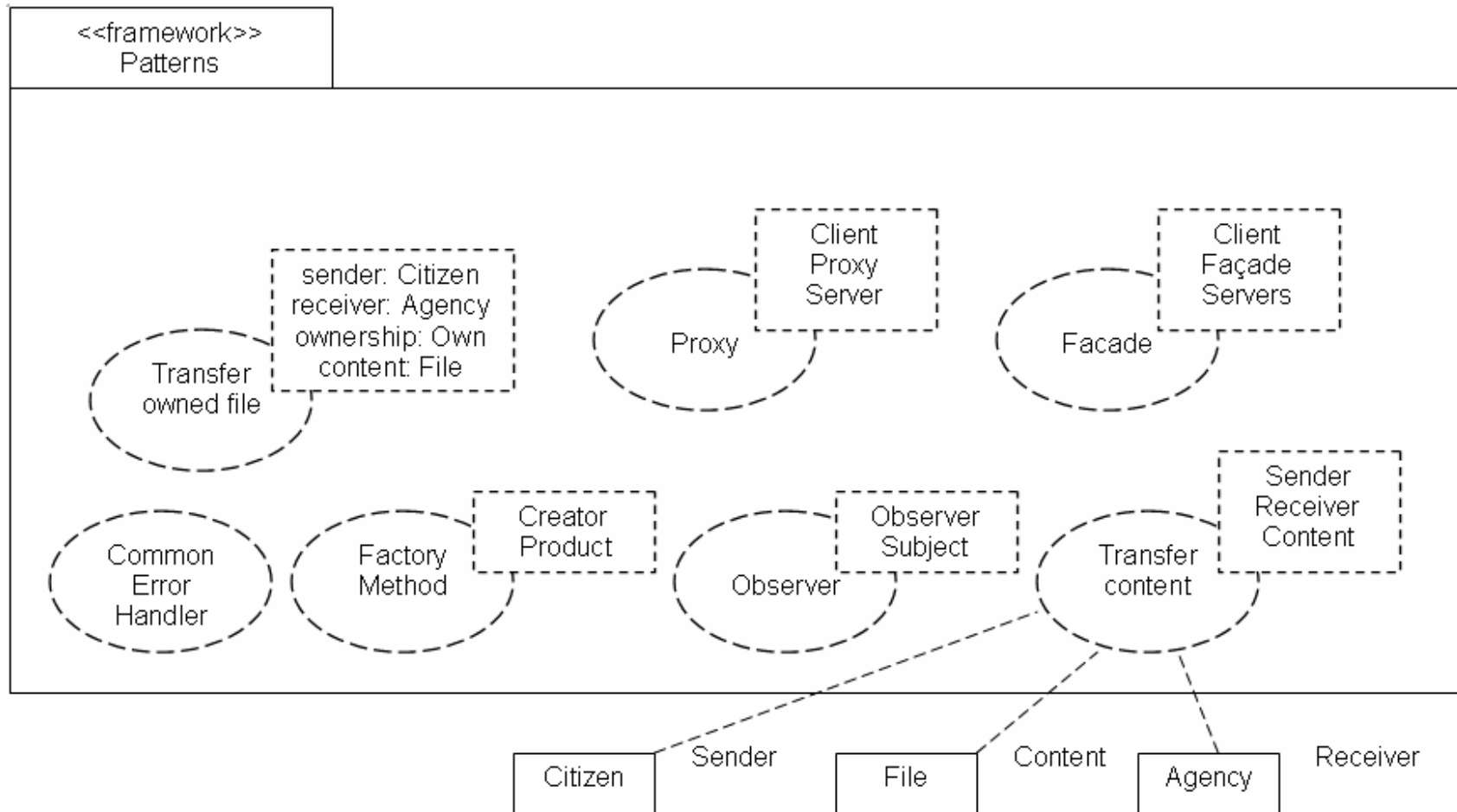
Frameworks are depicted as packages stereotyped with the “framework” keyword.

# Framework 2

---

- 1) frameworks are made up of a set of related classes that can be specialized or instantiated to implement an application
- 2) they lack the necessary application specific functionality and therefore not immediately useful
- 3) they may be seen as a prefabricated structure or template of a working application in which “plug-points” or “hot spots” are not implemented or are given overridable implementations
- 4) patterns can be used to document frameworks
- 5) they are physical realization of one or more patterns

# Example: Framework



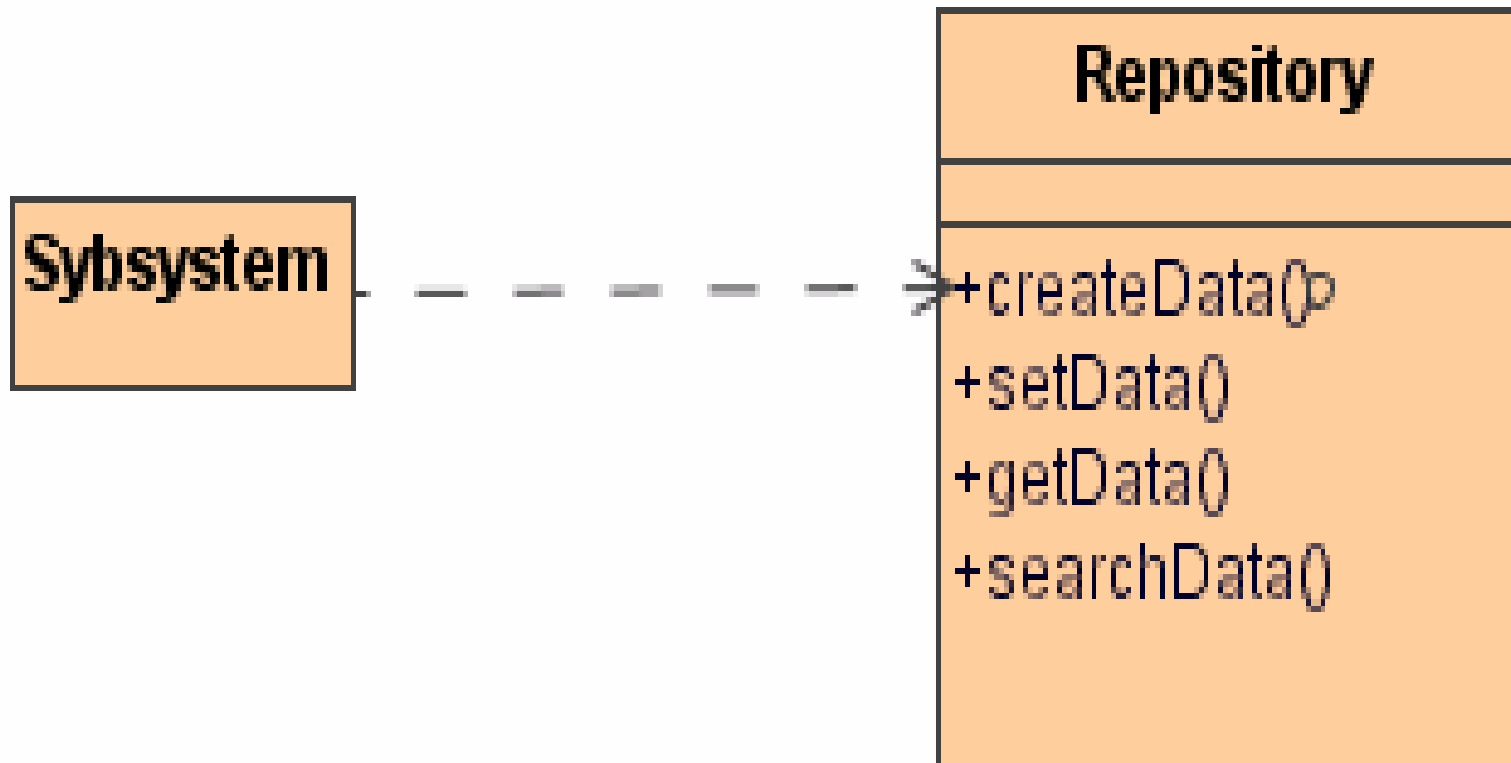
# Architecture Patterns/Styles

- 1) Repository
- 2) Model/View/Controller
- 3) Client-Server
- 4) Peer-to-Peer
- 5) Pipe-and-Filter

# Repository Architecture 1

- 1) subsystems access and modify data from a single data structure called the repository
- 2) subsystems are relatively independent and interact through the central data structure
- 3) control flow can be dictated either by the central repository or by the subsystem
- 4) it is usually employed in database applications, compilers and software development environment

# Repository Architecture 2



*UML Class Diagram Describing Repository Architecture*

# MVC Architecture 1

---

Subsystems are classified into three different types:

- a) **model** subsystems – maintains domain knowledge
- b) **view** subsystems – displays information to users
- c) **controller** subsystem – controls sequence of interaction

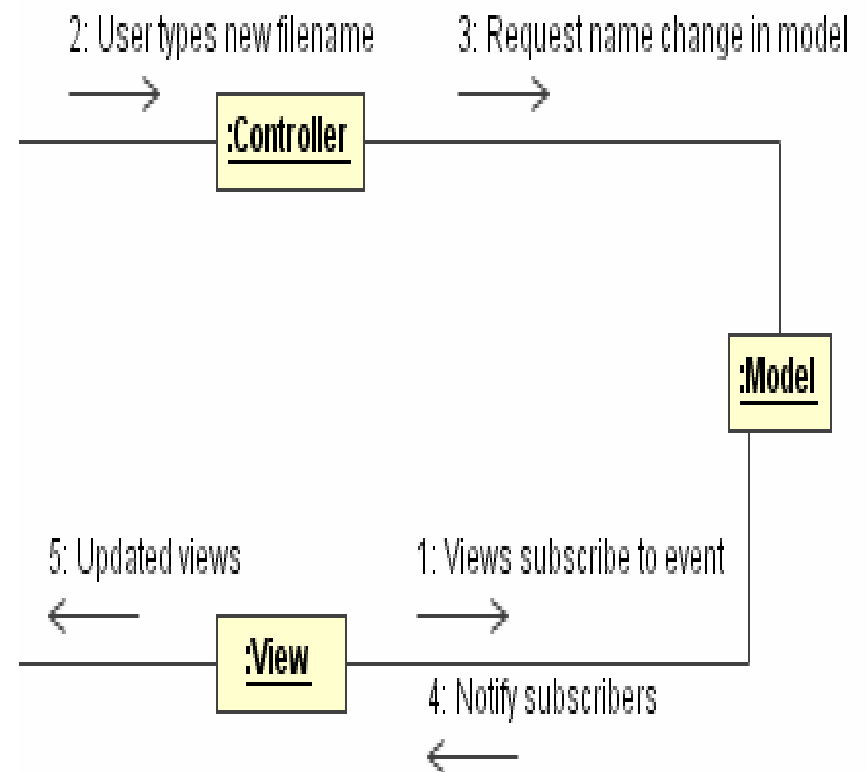
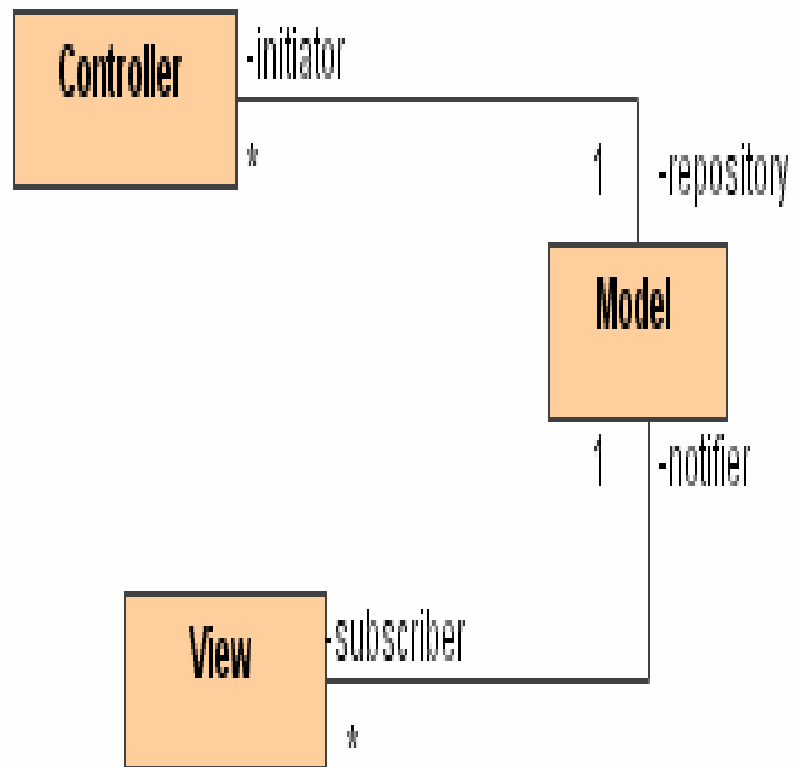
The Model subsystems are written independently of view or controller subsystems.

Changes in model's state are propagated to the view subsystem through the subscribe/notify protocol.

The MVC architecture is a special case of repository architecture; model is the central repository and the controller subsystem dictates control flow.



# MVC Architecture 2

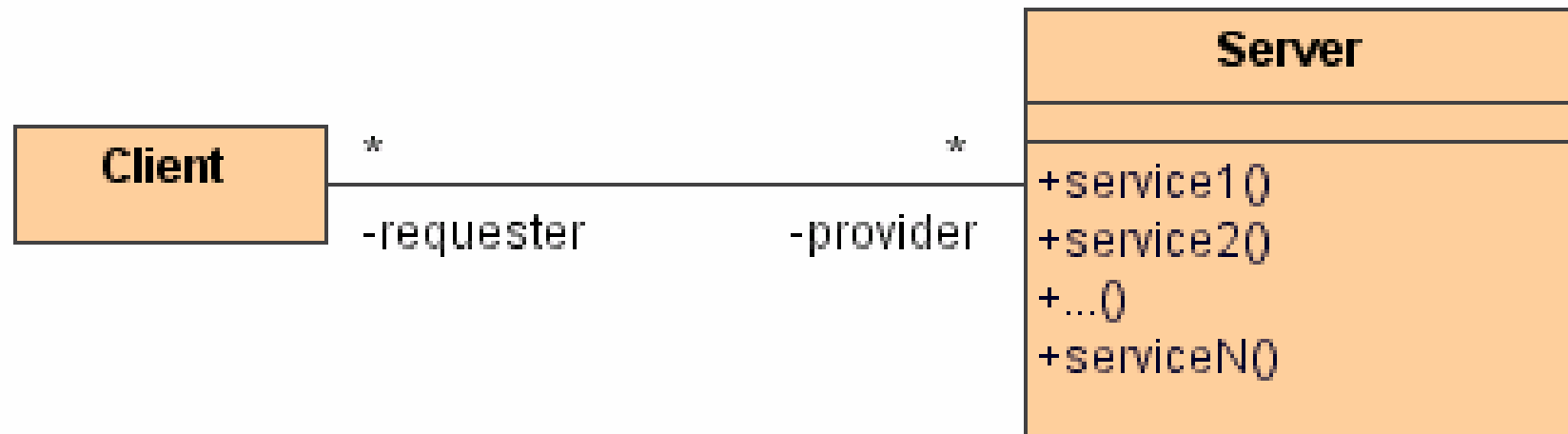


*Structural and Behavioural Description of MVC Architecture*

# Client – Server Architecture 1

- 1) there are two kinds of subsystems – the **Server** and **Client**
- 2) the Server subsystem provides services to instances of the other subsystems called clients; which interacts with the users
- 3) service requests are usually through remote procedure call or some other distributed programming techniques
- 4) control flow in clients and servers is independent except for synchronization to manage requests and receive results
- 5) there may be multiple servers subsystems like in the case of the world-wide web

# Client – Server Architecture 2

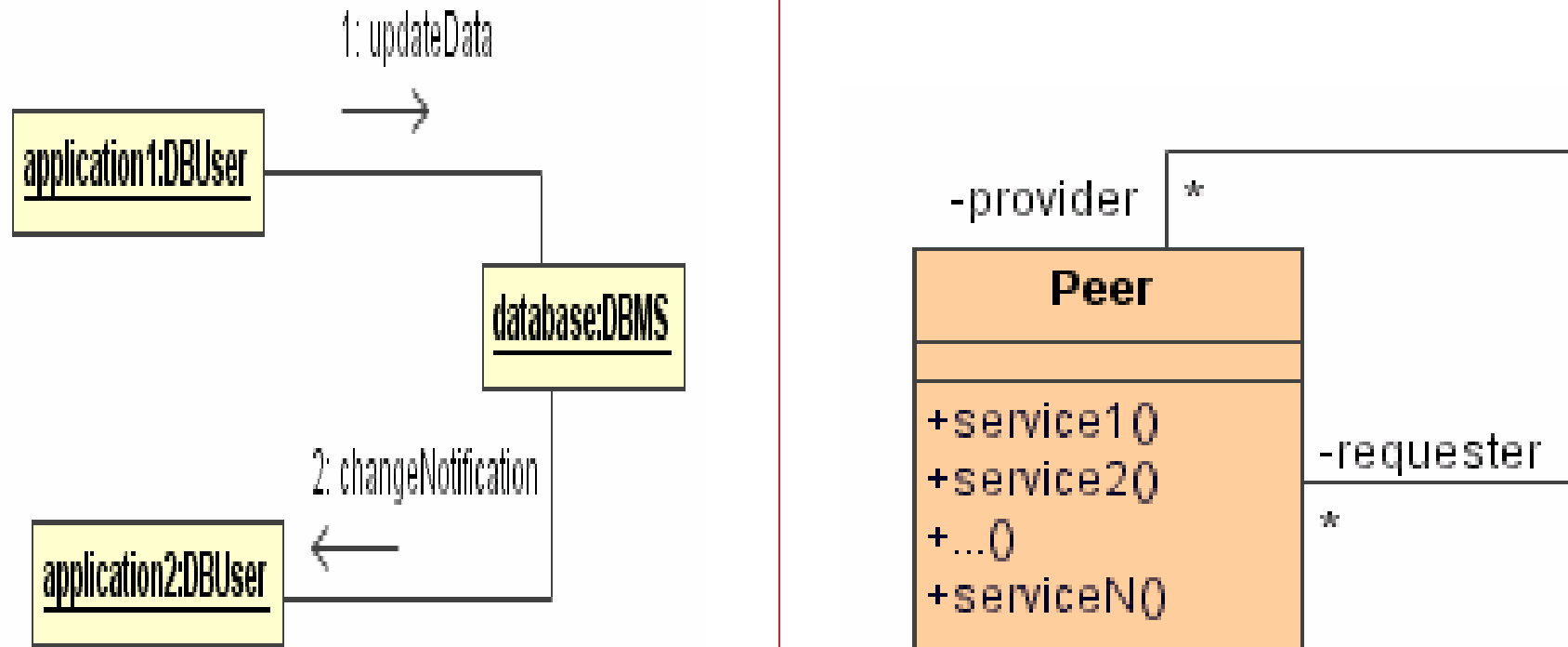


*Client – Server Architecture - Structural Description*

# Peer-to-Peer Architecture 1

- 1) a generalization of the client-server architecture in which subsystems can take up the roles of clients and servers dynamically
- 2) control flow within subsystems is independent from the others except for synchronization on requests
- 3) Peer-to-Peer architecture is more difficult to design than client server systems as there are possibilities of deadlocks

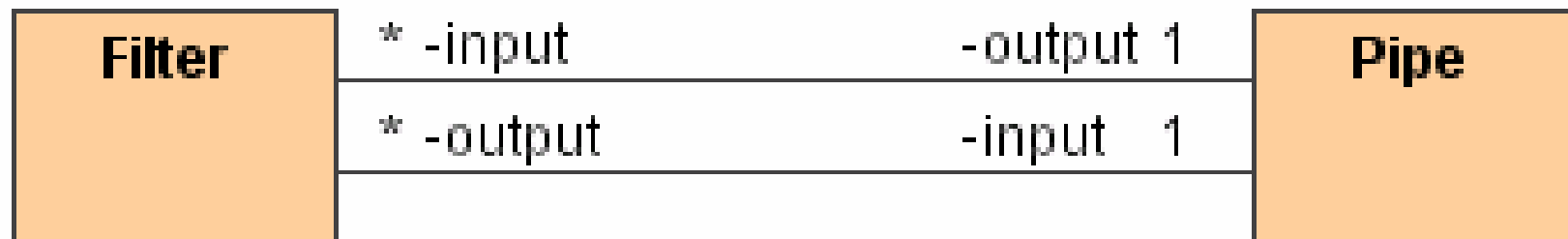
# Peer-to-Peer Architecture 2



# Pipe-and-Filter Architecture 1

- 1) subsystems process data received from a set of input and send results to other subsystems via set of outputs
- 2) subsystems are called **filters** and the association between filters are called **pipes**
- 3) filters only know the content and format of the data received on the input pipes and not the filters that produce them
- 4) each filter is executed concurrently and synchronization is done via the pipes
- 5) pipes and filters can be reconfigured as required

# Pipe-and-Filter Architecture 2



# System Operations Contract



# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary

# Contracts

---

## Definition

**Contracts** are constraints on a class that enable caller and callee to share the same assumption about class. The contract specifies constraints the caller must meet before using the class as well as the constraints that are ensured by the callee when used.

Contracts include there types of constraints:

- a) invariant
- b) precondition
- c) postcondition

Contracts are defined for important system operations.

# Contract: Invariant

## Definition

An **Invariant** is a predicate that is always true for all instances of a class. Invariant constraints are associated with classes or interfaces. They specify consistency conditions among class attributes.

**Contract C01:** *make\_application*

**Cross References:** use case - *submit application*

**Invariant:** *criminal record must be nil*

Applicant
-id_number : Integer -name -address [2] -internal_ref -age -criminal_record
+getName() +getId() make_application(id,age,type)

# Contract: Pre-Condition

## Definition

A **Pre-Condition** is a predicate that must be true before an operation is invoked. Pre-conditions are associated with specific operations and is a constraint on the caller.

**Contract C01:** *submit\_application*

**Cross References:** use case - *submit application*

**Precondition:** (1) *application has not been submitted earlier*; (2) *entry in application form is valid*

Application
-application_id -applicant_id -type_of_license -date_of_application -applicant_age -begin_date -end_date -application_submitted : boolean
+validate_entry() : boolean +submit_entry()

# Contract: Post-Condition

---

## Definition

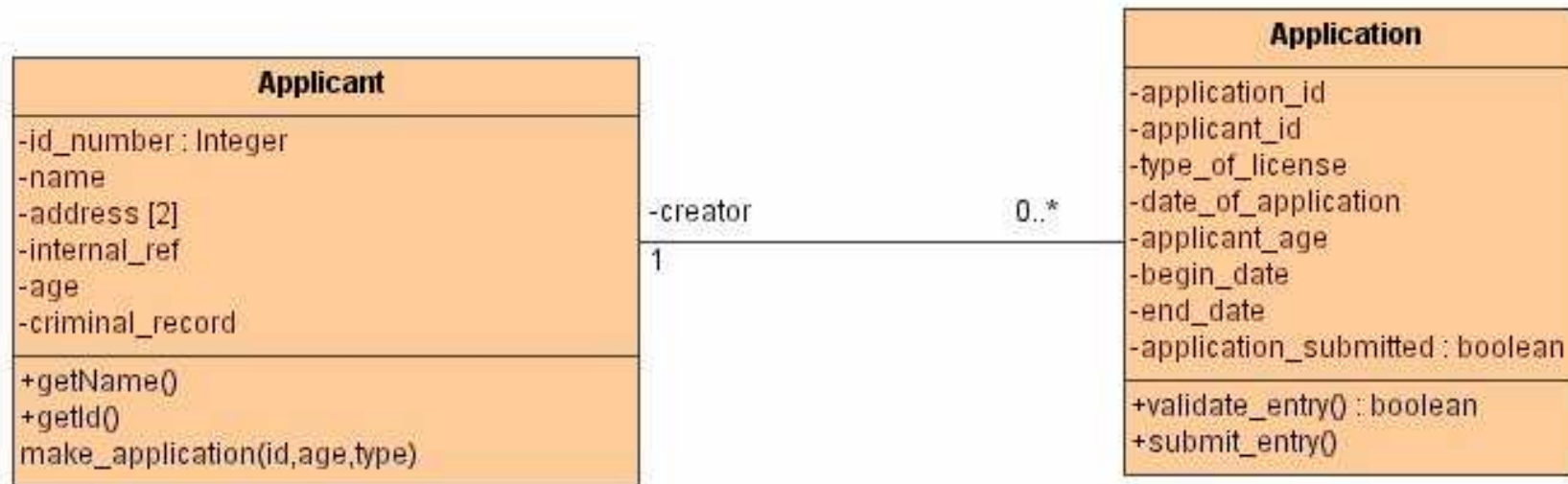
A **Post-Condition** is a predicate that must be true after an operation is invoked. Post-conditions are associated with specific operations and are used to specify the constraints that the object must ensure after execution of the operation.

**Contract C01:** *submit\_application*

**Cross References:** use case - *submit application*

**Post-condition:** (1) *a record of the application is created.*

# Example: Contract



make\_application Invariant  
no criminal record

submit\_entry Precondition  
application\_submitted has  
a value of FALSE  
AND  
validate\_entry()

submit\_entry Postcondition:  
application\_submitted has a  
value of TRUE

# GRASP Patterns

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary



# What is GRASP

---

General Responsibility Assignment Software Pattern

Responsibility:

- 1) a contract or an obligation of an object
- 2) responsibilities are related to the obligations of objects in terms of their behaviour

Two types:

- 1) doing responsibilities – actions that an object can perform
- 2) knowing responsibilities – knowledge that an object maintains

# GRASP – Responsibilities

## 1) doing

- a) doing something itself
- b) initiating an action or operation in other objects
- c) controlling and coordinating activities of other objects

## 2) knowing

- a) knowing about private encapsulated data
- b) knowing about related objects
- c) knowing about things it can derive or calculate

# GRASP Patterns

---

- 1) Information Expert
- 2) Creator
- 3) High Cohesion
- 4) Low Coupling
- 5) Controller

# Expert 1

---

**Pattern Name** : Expert

**Solution** : assign a responsibility to the information expert – the class that has information necessary to fulfill the responsibility.

**Problem** : what is the most basic principles by which responsibilities are assigned in OOD?

**Example:** consider the “Applicant” and “Application” classes in the contract example. Which class should be responsible for submitting an application?

Since the Application class possesses the required information necessary for submitting an application (basic information, begin application and end application dates etc.), we delegate this responsibility to it.

# Expert 2

---

- 1) **Expert pattern** is the most applied GRASP pattern in OOD, supporting the common intuition of objects do things related to the information they have
- 2) fulfillment of responsibility may require information spread across different classes of objects, indicating the possibilities of many partial experts who will collaborate in the task
- 3) Expert patterns help maintain encapsulation since objects use their own information to fulfill responsibilities

# Creator 1

---

**Pattern Name** : Creator

**Solution** : assigns class B the responsibility to create an instance of class A (B is a creator of A objects) if one of the following is true:

- B aggregates A objects
- B contains A objects
- B records instances of A objects
- B closely uses A objects
- B has initializing data that will be passed to A when it is created (thus B is an expert w.r.t. creating A objects)

**Problem**: what should be responsible for creating a new instance of some class?

# Creator 2

---

**Example:** still consider the contract example involving the Applicant and Application Class. Obviously the Applicant class closely uses the Application objects. In fact, the *Applicant* possess vital initializing data for the *Application* objects and thus an expert at creating *Application* objects.

So we can assign the Applicant class the responsibility of creating an instance of the Application Class.

Thus the inclusion of the *make\_application()* method in the Applicant class

# Low Coupling 1

---

**Coupling** is a a measure of how strongly one class is connected to, has knowledge of, or relies upon other classes. A class with low or weak coupling is not dependent on too many class.

**Pattern Name:** Low Coupling

**Solution:** assign a responsibility so that coupling remains low.

**Problem:** how to support low dependency and increased reuse?



# Low Coupling 2

---

**Example:** consider the MVC architecture described earlier, where the controller component will dispatch requests to the model component.

If the model component was implemented as individual classes which will be called by the controller component, then there will be a lot of dependencies involved the architecture of the system.

On the other hand, we can conceive of a “broker or façade” class which takes the request from the controller and identifies appropriate classes to service the request.

# Low Coupling 3

---

- 1) common forms of coupling from class X to class Y:
  - a) class X has an attribute that refers to a class Y itself
  - b) class X has a method which references an instance of class Y, or Class Y itself, by an means (parameters or local variable of type Class Y, or the object returned from a message being an instance of class Y
  - c) class X is a direct or indirect subclass of class Y
- 2) coupling may not be too important if reuse is not a goal
- 3) no absolute measure of when coupling is too high
- 4) very little or no coupling between classes is rare and not interesting

# High Cohesion 1

---

**Cohesion** is a measure of how strongly related and focused the responsibilities of a class are.

A class with high cohesion has highly related functional responsibilities and does minimal amount of work.

**Pattern Name:** High Cohesion

**Solution:** assign a responsibility so that cohesion remains high

**Problem:** how to keep complexity manageable?

# High Cohesion 2

---

**Example:** again consider the contract example. Since applicants make applications one may ask if the “Applicant” and the *Application* classes could be merged.

This would require the *Applicant* class to bear the responsibilities of *making*, *validating* and *submitting* an *application*.

This strategy strongly ties the concept of an *Application* with an *Applicant*.

# Controller 1

---

A **controller** is a non-user interface object responsible for receiving or handling a system event. A controller defines the method for the systems operations.

**Pattern Name:** Controller

**Solution:** assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- a) the overall system, device, or subsystem (façade controller)
- b) a use case scenario within which the system events occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator or <UseCaseName>Session

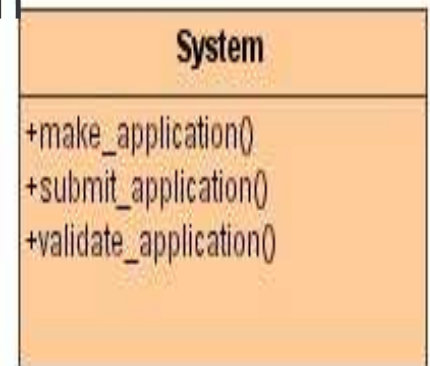
# Controller 2

---

**Problem:** Who should be responsible for handling an input system event?

An input system event is an event generated by an external actor. They are associated with system operations which respond to system events.

**Example:** consider the system operations shown in the System class. We may ask who should be the controller for system events such as *make\_application* and *submit\_application*?



By the controller pattern, we have some choices:

- (1) the overall *LicenseOnline* system
- (2) a receiver or handler of all system events of a use case scenario (*SubmitApplicationHandler*)

# Architecture Model for Case Study

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

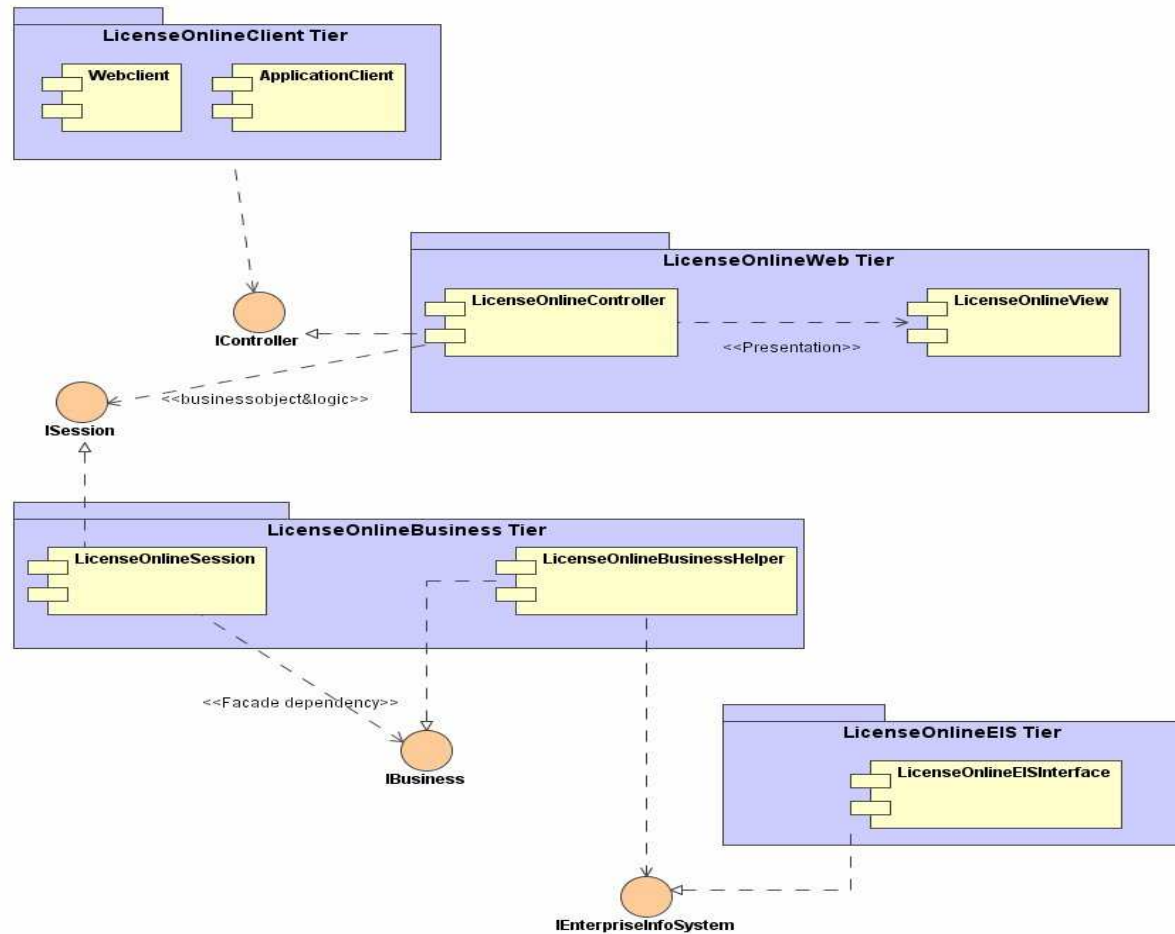
7) GRASP Patterns

8) Architecture Model for Case Study

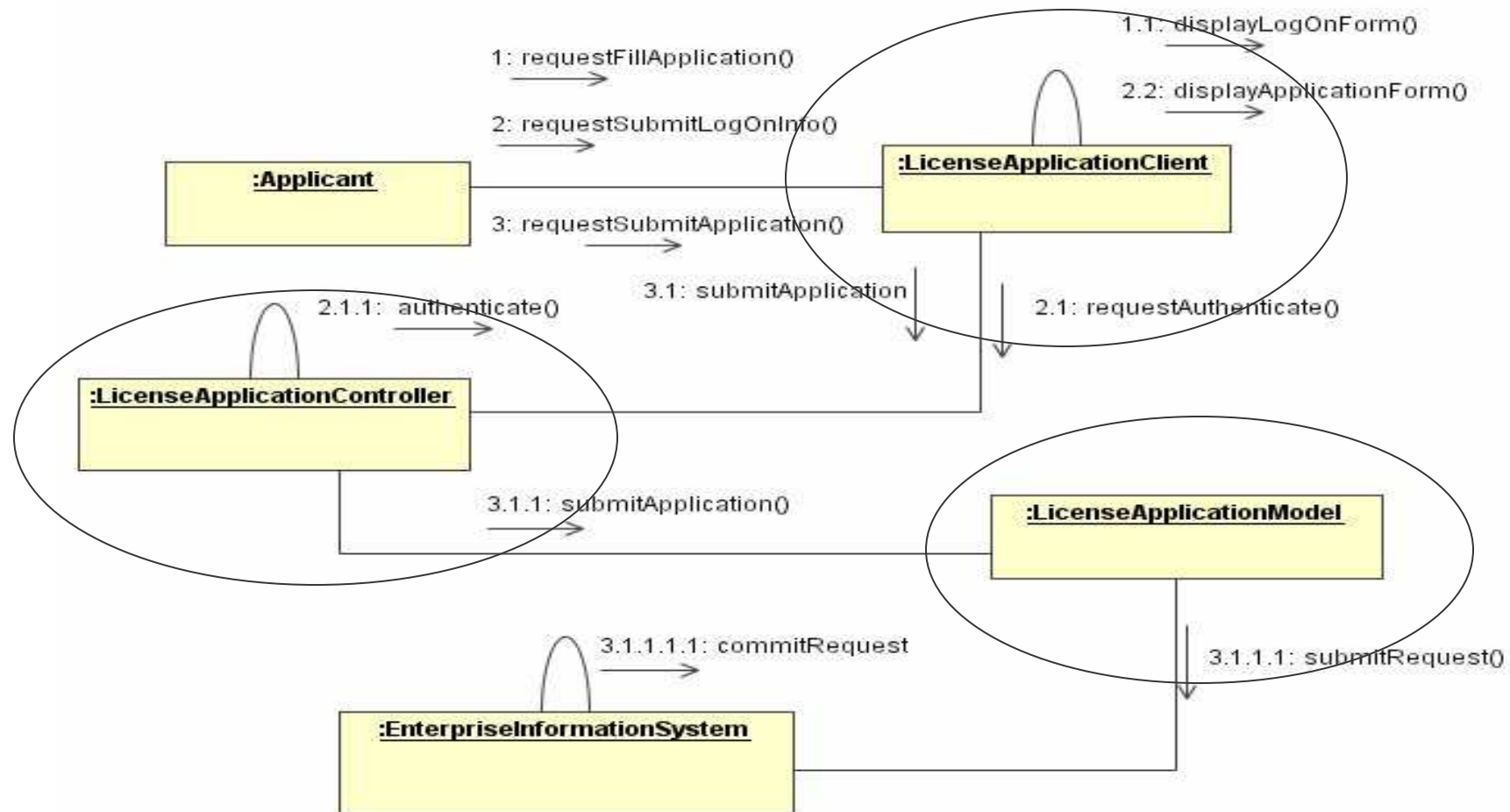
9) Summary



# Architecture: Structural Model



# Architecture: Behavioural Model



# Summary

# Architecture Modelling

---

1) Software Architecture Concepts

2) Packages

3) Collaboration Diagrams

4) Component Diagrams

5) Architectural Patterns

6) System Operations Contract

7) GRASP Patterns

8) Architecture Model for Case Study

9) Summary

# Summary 1

---

Architecture is concerned with the structural organization of system components as well as how they interact to provide the system's overall behaviour or functionality.

Packages are general purpose mechanisms for organizing modelling elements into groups.

Well structured packages must be loosely coupled and very cohesive.

Five stereotypes may be applied to packages: façade, framework, stub, subsystem and system.

# Summary 2

---

A collaboration is a society of classes which provides some cooperative behaviour that is more than the sum of all its parts.

Collaborations have both structural and behavioural aspects

Collaborations may realize a use case or an operation.

A collaboration diagram shows interactions organized around the structure of a model, using either classes and associations or instances and links.

# Summary 3

---

A component is a physical, replaceable part that conforms to and provides the realization of a set of interfaces.

An interface is a collection of operation that are used to specify a service of class or components.

There are three categories of components: deployment, work product and execution.

Components may be stereotyped as executable, library, table, file or document.

# Summary 4

---

A component diagram consists of specifying classes, implementing artifacts, components, interfaces and relationships between these model elements.

Basic architectural patterns or styles include: repository; MVC; client-server; peer-to-peer; and pipe-and-filter.

The General Responsibility Assignment Software Pattern provides five basic patterns: Information Expert, Creator, High Cohesion, Low Coupling and Controller.



# Exercise – Project 1

---

- 1) Describe a high level architecture of your system using packages.
- 2) List the various classifiers and associations that are essential in presenting a collaboration diagram for your system.
- 3) Present the above elements in a specification level collaboration diagram for your system.
- 4) Identity the major components of your system and relate these components to their specifying classes and defining interfaces.
- 5) Using the same specification level collaboration diagram drawn in 3, show a conforming instance level collaboration diagram that supports a particular use case of your system.

# Exercise – Project 2

---

- 6) Consider the architectural patterns discussed earlier, justify your choice of a suitable architecture pattern.
- 7) Discuss the tradeoffs associated with your system's architectural style.
- 8) Highlight some principles that underpin responsibility assignment in your system's architecture.

# Design Modelling

# Overview

---

1) The Course

2) Object-Oriented Concepts

3) UML Basics

4) Case Study

5) Modelling:

a) Requirements

b) Architecture

c) Design

d) Implementation

e) Deployment

6) UML and Unified Process

7) Tools

8) Summary

# Design Concepts

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

# What is System Design?

*"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies"*

C.A.R. Hoare

## Definition

**System design** is the transformation of analysis models of the problem space into design models (based on the solution space). It involves selecting strategies for building the system e.g. software/hardware platform on which the system will run and the persistent data strategy.

# System Design - Overview

## 1) Analysis produces:

- a) a set of non-functional requirements and constraints
- b) a use case model describing functional requirements
- c) an object model, describing entities
- d) a sequence diagram for each use case showing the sequence of interaction among objects

## 2) Design results in:

- a) a list of design goals – qualities of the system
- b) software architecture describing the subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, major policy decision such as control flow, access control, and data storage



# System Design – Activities

- 1) definition of design goals
- 2) decomposition of system into sub-system
- 3) selection of off-the-shelf and legacy components
- 4) mapping of sub-systems to hardware
- 5) selection of persistent data management infrastructure
- 6) selection of access control policy
- 7) selection of a global control flow mechanism
- 8) handling of boundary conditions

# Some Design Activities 1

- 1) Define goals:
  - a) derived from non-functional requirements
- 2) Hardware and software mapping:
  - a) what computers (nodes) will be used?
  - b) which node does what?
  - c) how do nodes communicate?
  - d) what existing software will be used and how?
- 3) Data management:
  - a) which data needs to be persistent?
  - b) where should persistent data be stored?
  - c) how will the data be stored?

# Some Design Activities 2

## 4) Access control:

- a) who can access which data?
- b) can access control be changed within the system?
- c) how is access control specified and realized?

## 5) Control flow:

- a) how does the system sequence operations?
- b) is the system event-driven?
- c) does it need to handle more than one user interaction at a time?

# What is Object Design?

- 1) system design describes the system in terms of its architecture, such as its subsystem decomposition, its global control flow and its persistency management
- 2) object design includes:
  - a) **service specification** – precise description of class interface
  - b) **component selection** – identification of off-the-shelf components and solution objects
  - c) **object model restructuring** – transform the object design model to improve understandability and extensibility
  - d) **object model optimization** – transform object model to address performance (response time, memory etc.)

# Object Design Activities 1

## 1) specification

- a) missing attributes and operations
- b) type signatures and visibility
- c) constraints
- d) exceptions

## 2) component selection

- a) adjusting class libraries
- b) adjusting application frameworks

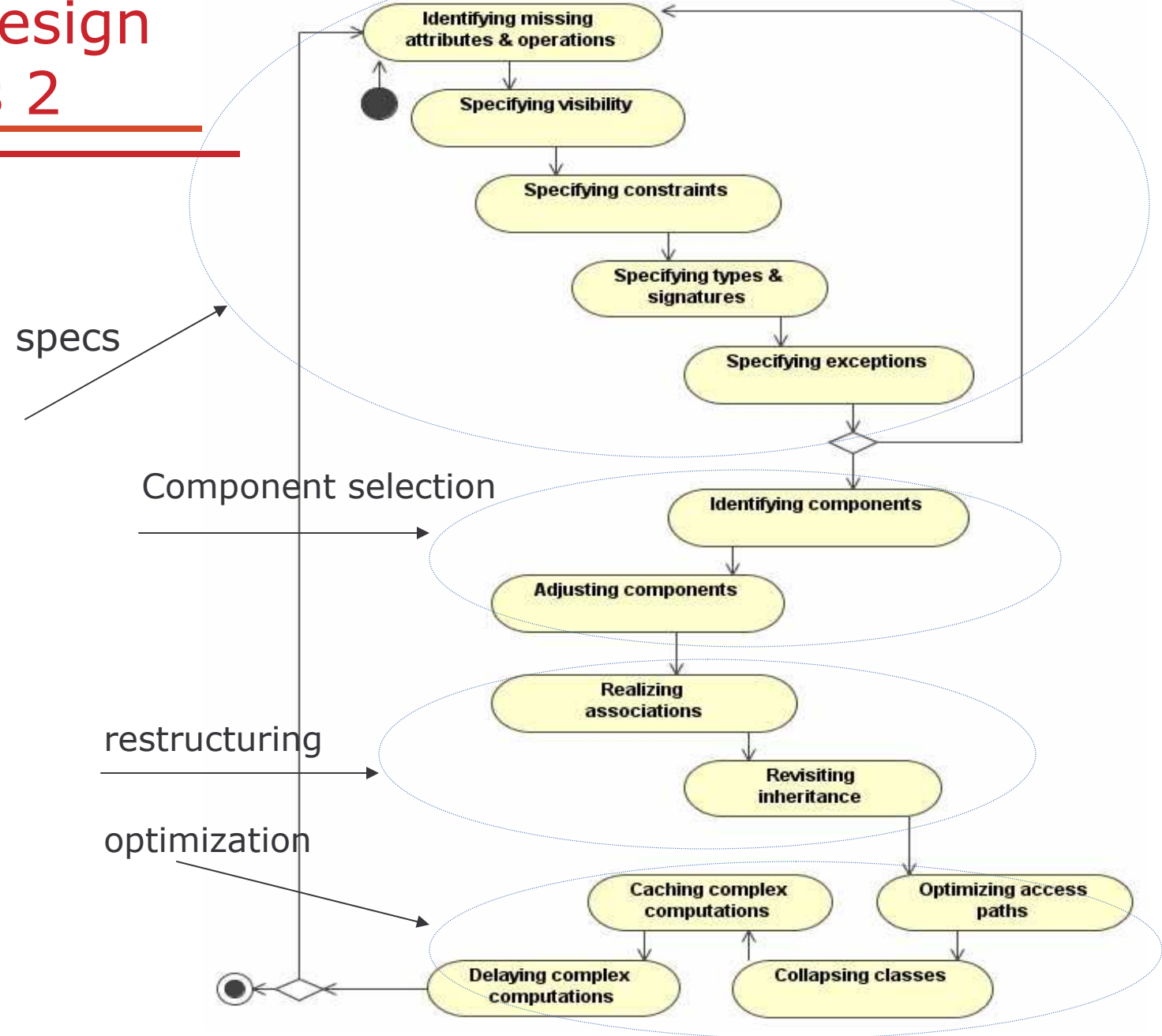
## 3) restructuring

- a) realizing associations
- b) increasing reuse
- c) removing implementation dependencies

## 4) optimization

- a) access paths
- b) collapsing objects
- c) caching results of expensive components
- d) delaying expensive components

# Object Design Activities 2



# Design Pattern

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary



# Design Patterns 1

---

## Definition

**Design patterns** are partial solutions to common problems. They name, abstract, and identify the key aspects of common design structure that make them useful for creating reusable object-oriented design.

Design patterns are composed of small number of classes that through **delegation** and **inheritance**, provide **robust** and **modifiable** solutions.

## Some common problems:

- a) separating interfaces from a number of alternate implementations
- b) wrapping around a set of legacy classes
- c) protecting a caller from changes associated with specific platforms

# Design Patterns 2

---

- 1) design patterns capture expert knowledge and design tradeoffs and support the sharing of architectural knowledge among developers
- 2) design patterns as a shared vocabulary can clearly document the software architecture of a system
- 3) allow design engineers relate to one another at a higher level of abstraction

# Design Patterns: Description 1

Attribute	Description
Pattern Name and Classification	Conveys the essence of the pattern succinctly.
Intent	A short statement that answers the following questions: what does the design do? Its rationale and intent.
Also Known As	Other well known names for the patterns, if any
Motivation	A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.
Applicability	What are the situations in which the design pattern can be applied?
Structure	Class diagram and sequence diagram for the classes involved in the pattern.
Participants	The classes and/or the object participating in the design pattern and their responsibilities.
Collaborations	How participant objects collaborate to carry out their responsibilities.

# Design Patterns: Description 2

Attribute	Description
Consequences	How does the pattern support its objectives? What are the tradeoffs?
Implementation	What pitfalls, hints, or techniques you should be aware of when implementing the pattern.
Sample Code	Code fragment to show implementation.
Known uses	Example of patterns found in real systems.
Related Patterns	Which design patterns are closely related to this one?

# Design Patterns Catalog

---

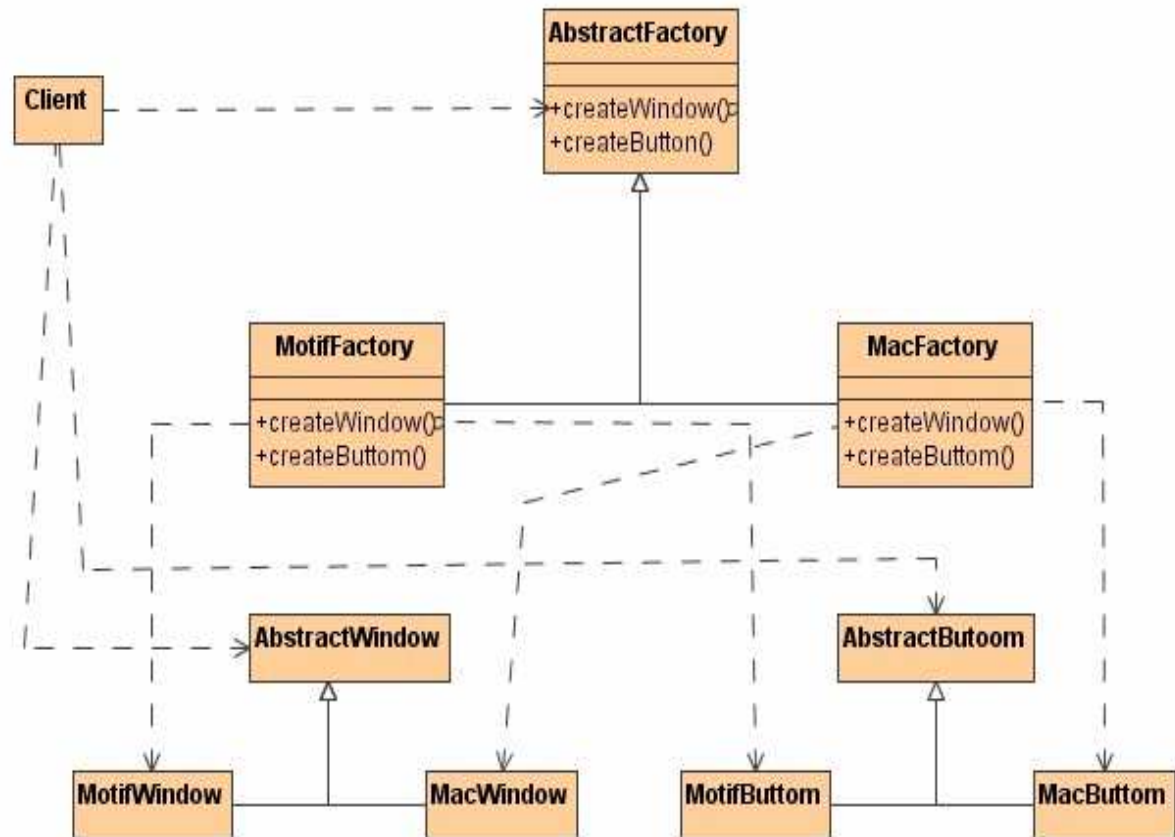
1.	Abstract Factory (C)	13.	Interpreter (B)*
2.	Adapter (S), Adapter (S)*	14.	Iterator (B)
3.	Bridge (S)	15.	Mediator (B)
4.	Builder (C)	16.	Memento (B)
5.	Chains of Responsibility (B)	17.	Observer
6.	Command (B)	18.	Prototype (C)
7.	Composite (S)	19.	Proxy (S)
8.	Decorator (S)	20.	Singleton (C)
9.	Façade (S)	21.	State (B)
10.	Factory Method (C)*	22.	Strategy (B)
11.	Flyweight (S)	23.	Template Method (B)*
12.	Visitor (B)		

*C – creational pattern, S – structural pattern, B – Behavioural pattern, \* - Class Scope*

# Applying Abstract Factory

Problem:  
encapsulating  
platforms

Solution:  
abstract factory  
provides an  
interface for  
creating families  
of related or  
dependent  
objects without  
specifying their  
concrete classes



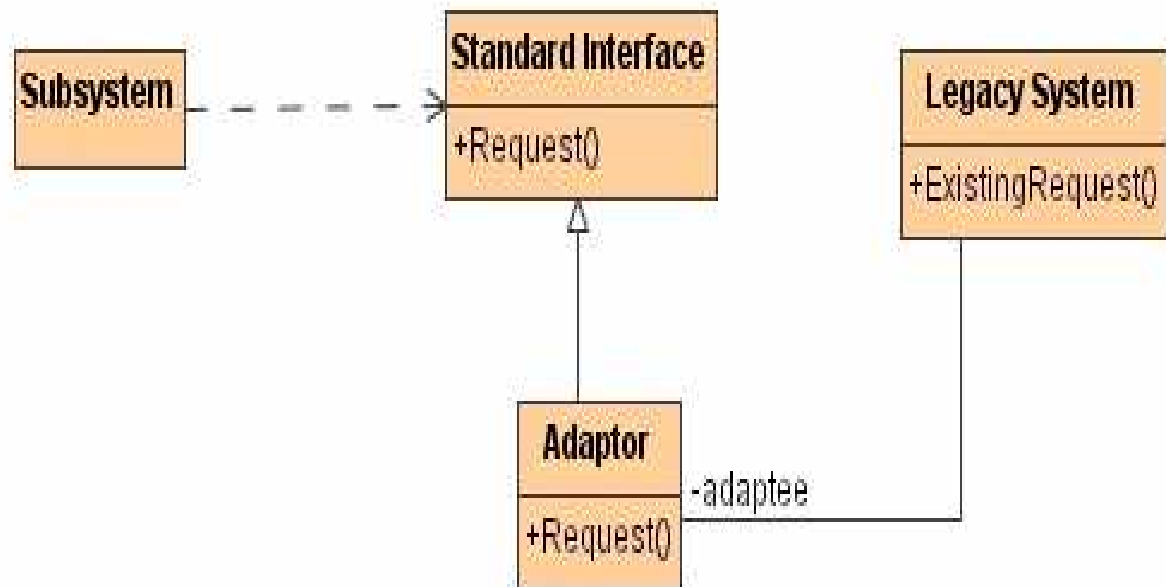
# Applying Adapter

Problem:

wrapping  
around legacy  
code

Solution:

adapter  
encapsulates a  
piece of legacy  
code not  
designed to  
work with the  
system



# Applying Bridge

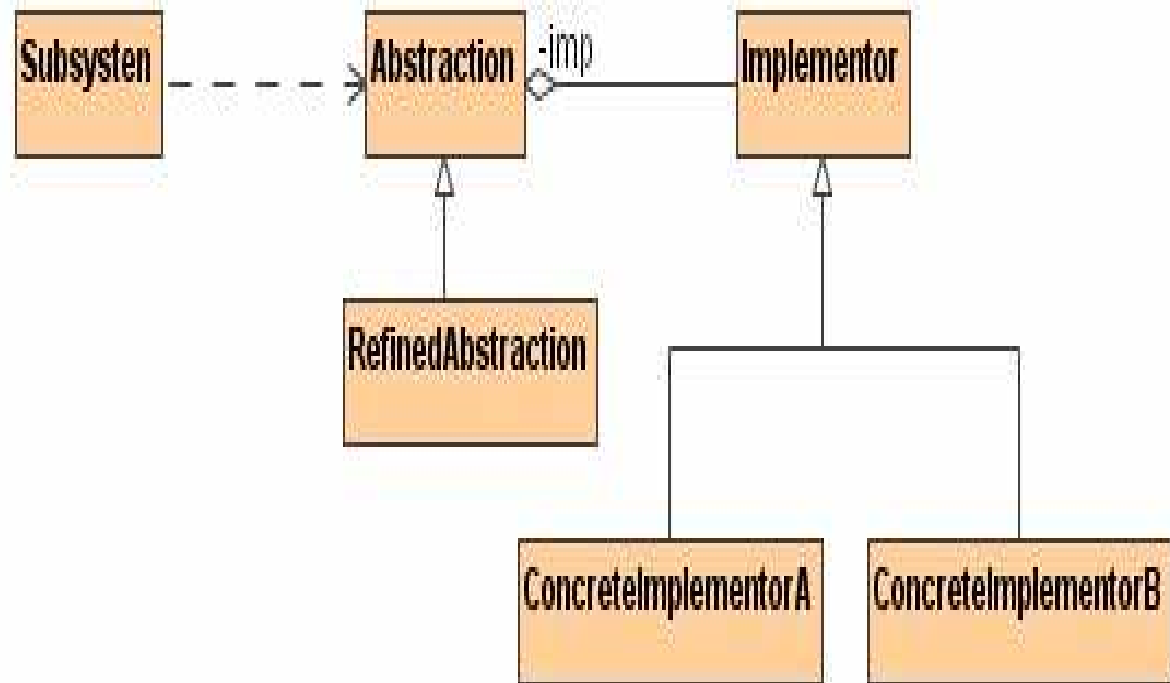
Problem:

allowing for  
alternate  
implementation

Solution:

**bridge**

decouples the  
interface from  
its  
implementation

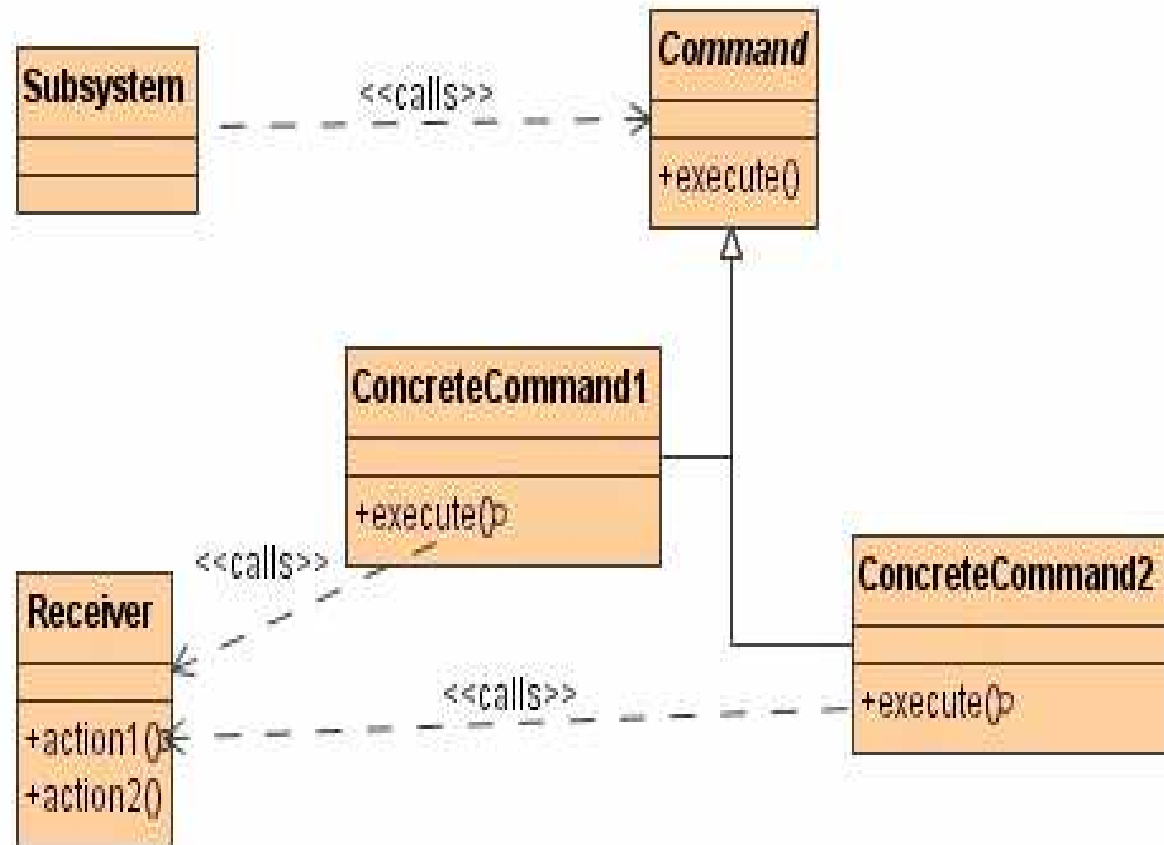




# Applying Command

Problem:  
encapsulating  
control

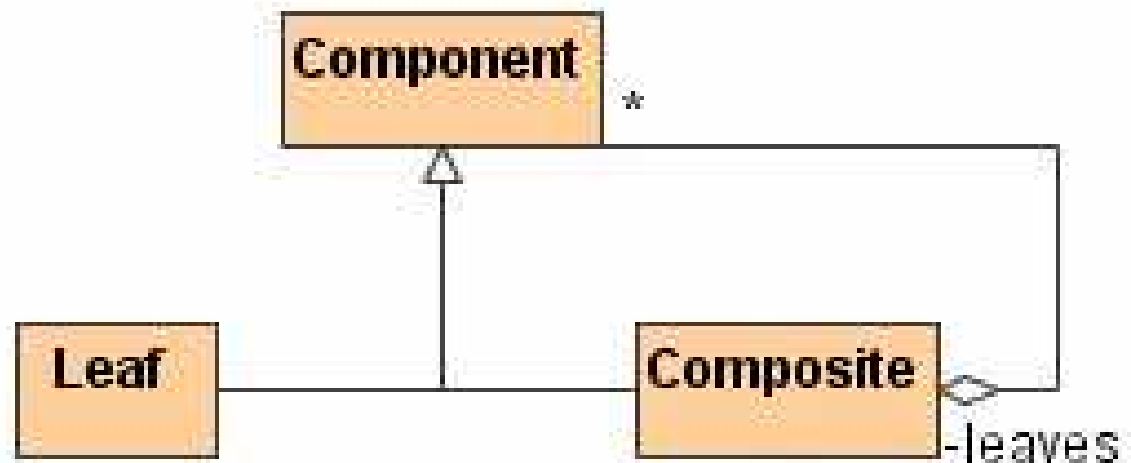
Solution:  
**command**  
encapsulates a  
control such  
that user  
requests can be  
treated  
uniformly



# Applying Composite

Problem:  
representing  
recursive  
hierarchies

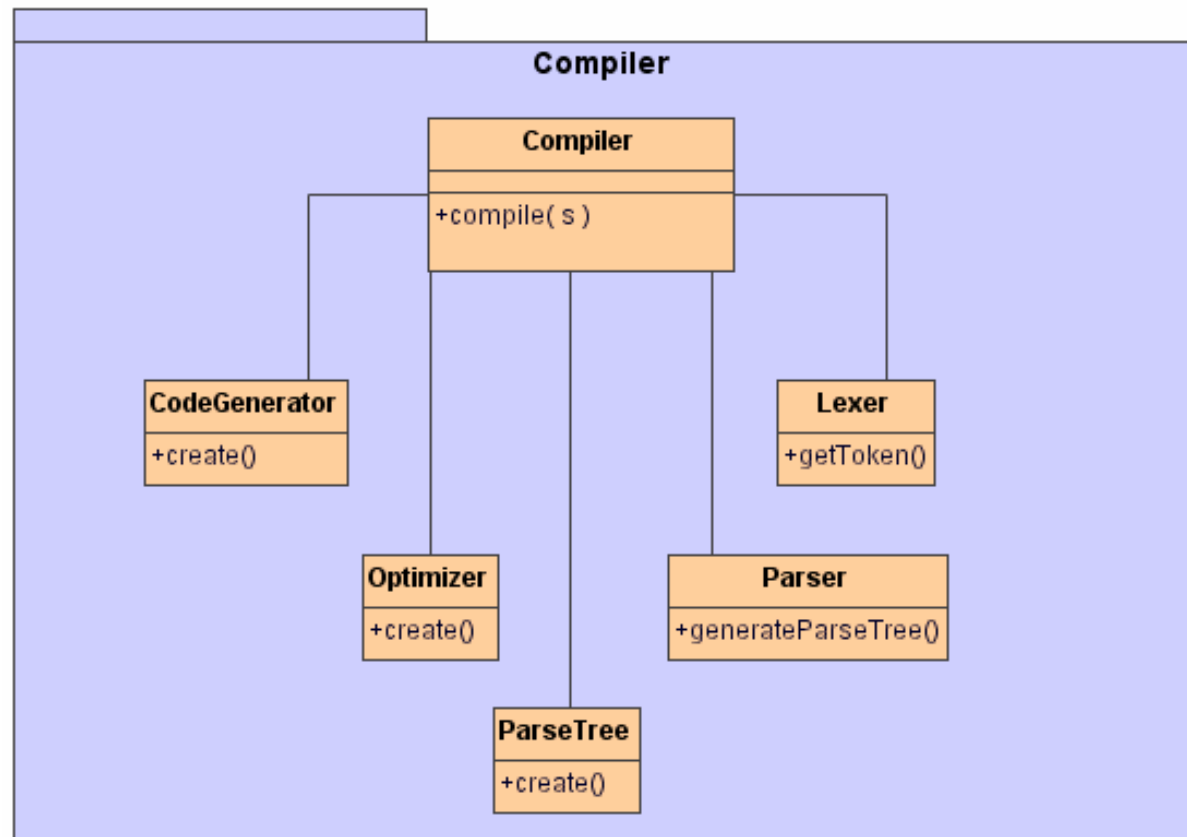
Solution:  
**composite**  
represent  
recursive  
hierarchies



# Applying Façade

Problem:  
encapsulating  
subsystems

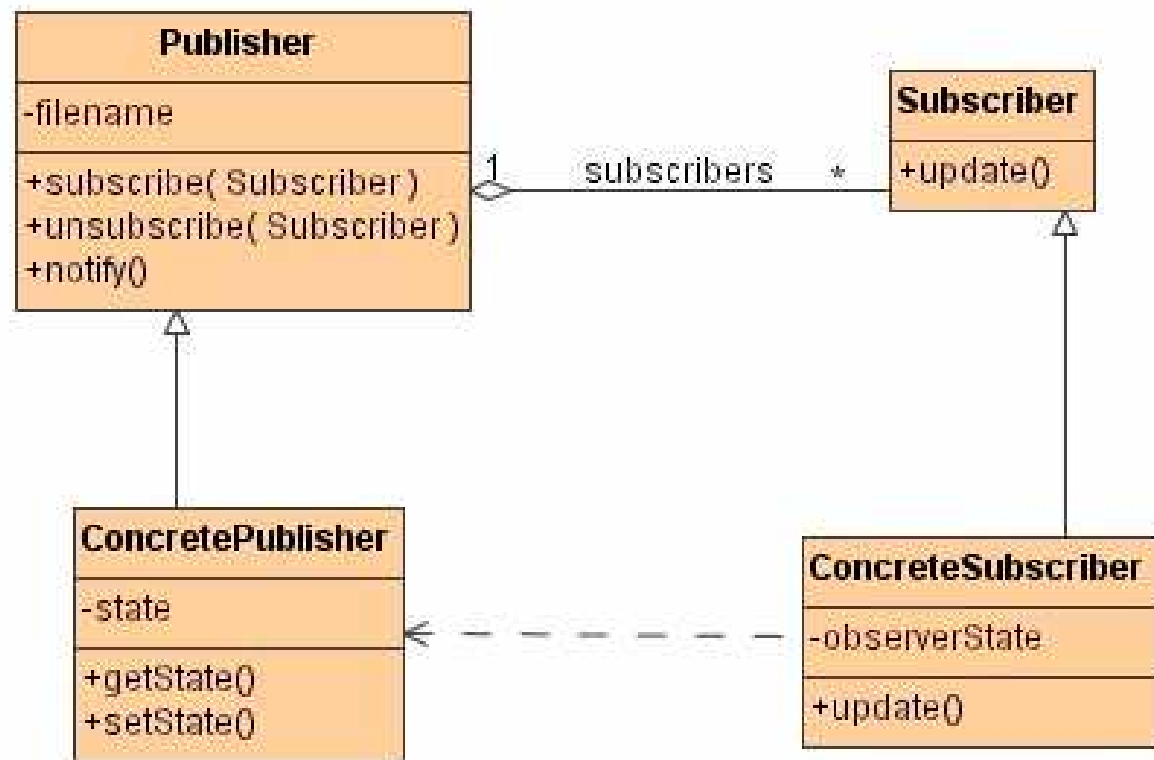
Solution:  
**façade** reduces  
dependencies  
among classes  
by encapsulating  
subsystems from  
with simple  
unified interfaces



# Applying Observer

Problem:  
decoupling  
entities from  
view

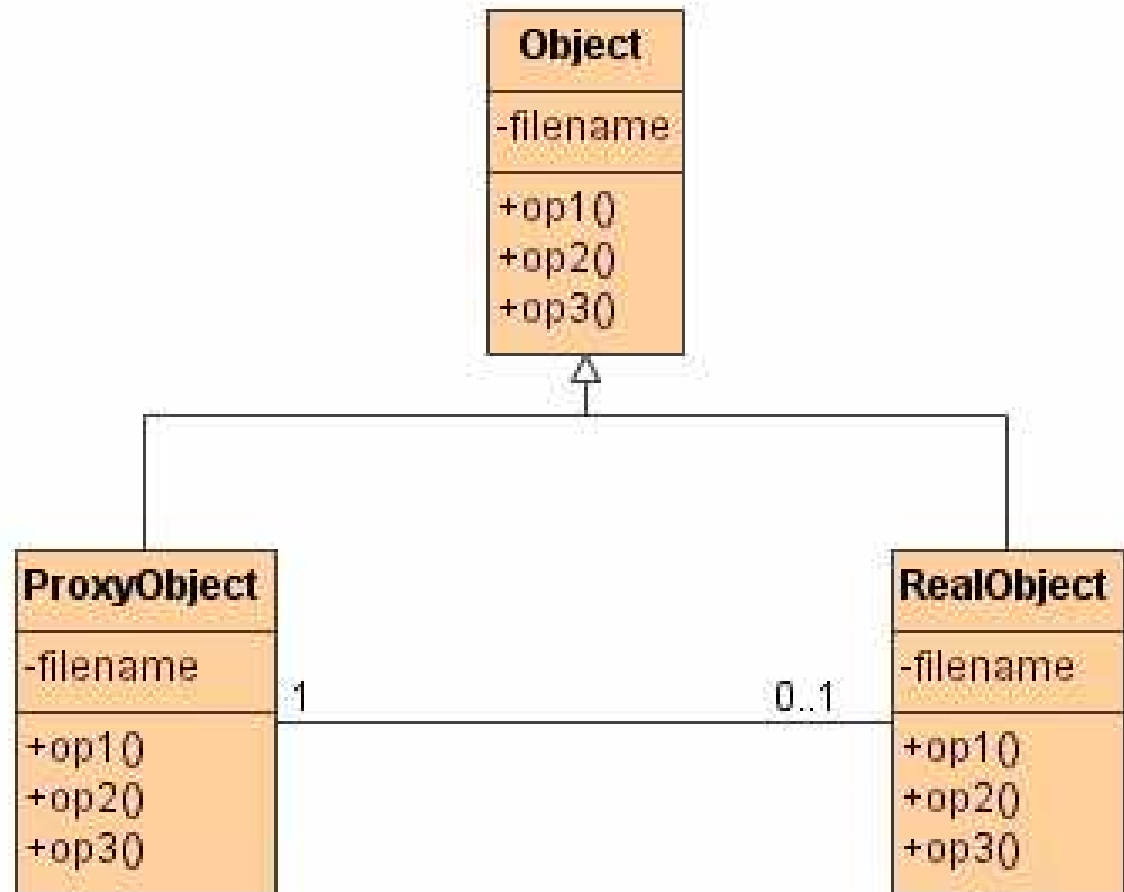
Solution:  
**observer** allows  
us to maintain  
consistency  
across the state  
of one publisher  
and many  
subscribers



# Applying Proxy

Problem:  
encapsulating  
expensive  
objects

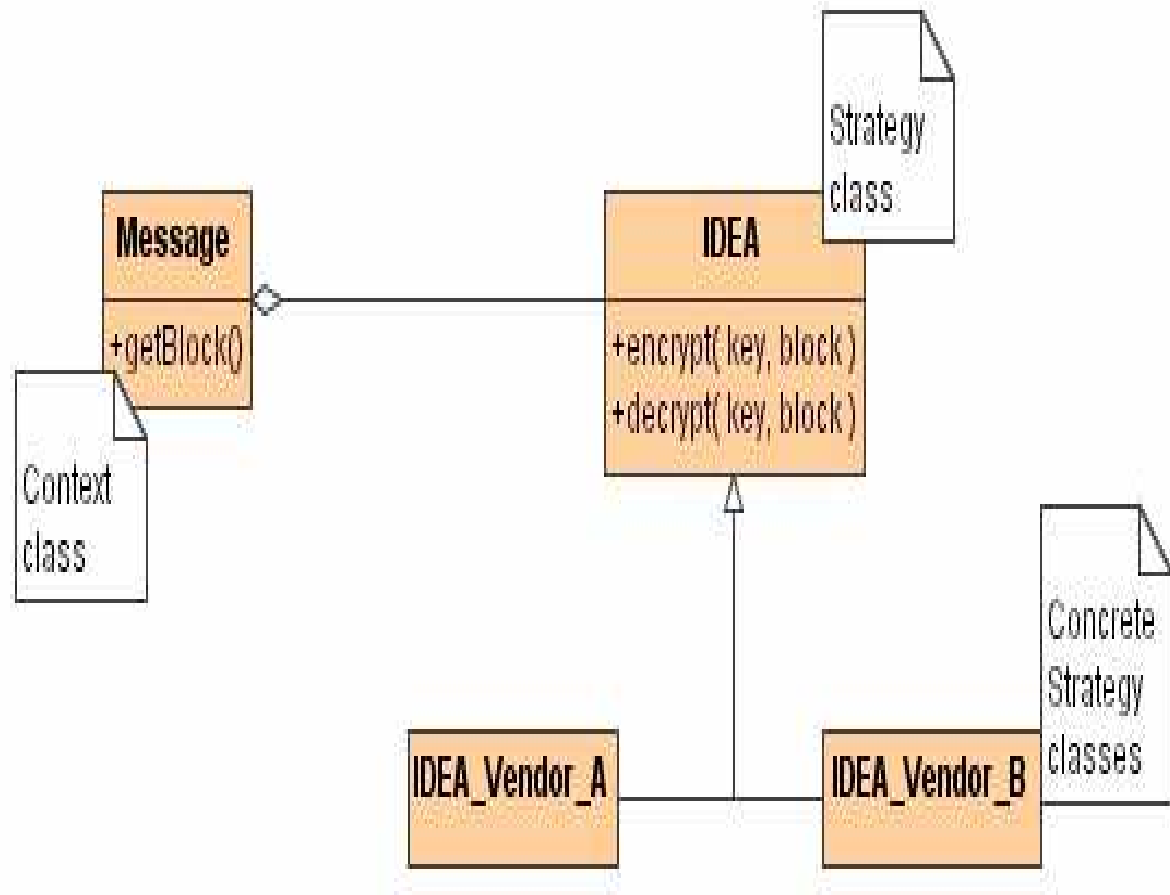
Solution:  
**proxy** improves  
the performance  
or the security of  
a system by  
delaying  
expensive  
computations, ...  
until when  
needed



# Applying Strategy

Problem:  
encapsulating  
algorithms

Solution:  
**strategy**  
decouples an  
algorithm from  
its  
implementations



# Design Class Diagrams

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary



# Design Class Diagram

A **Design Class Diagram** (DCD) provides the specification for software classes and interfaces in an application.

Typical information contained in a DCD includes:

- 1) classes, associations and attributes
- 2) interfaces, with their operations and constants
- 3) methods
- 4) attributes type information
- 5) navigability
- 6) dependencies

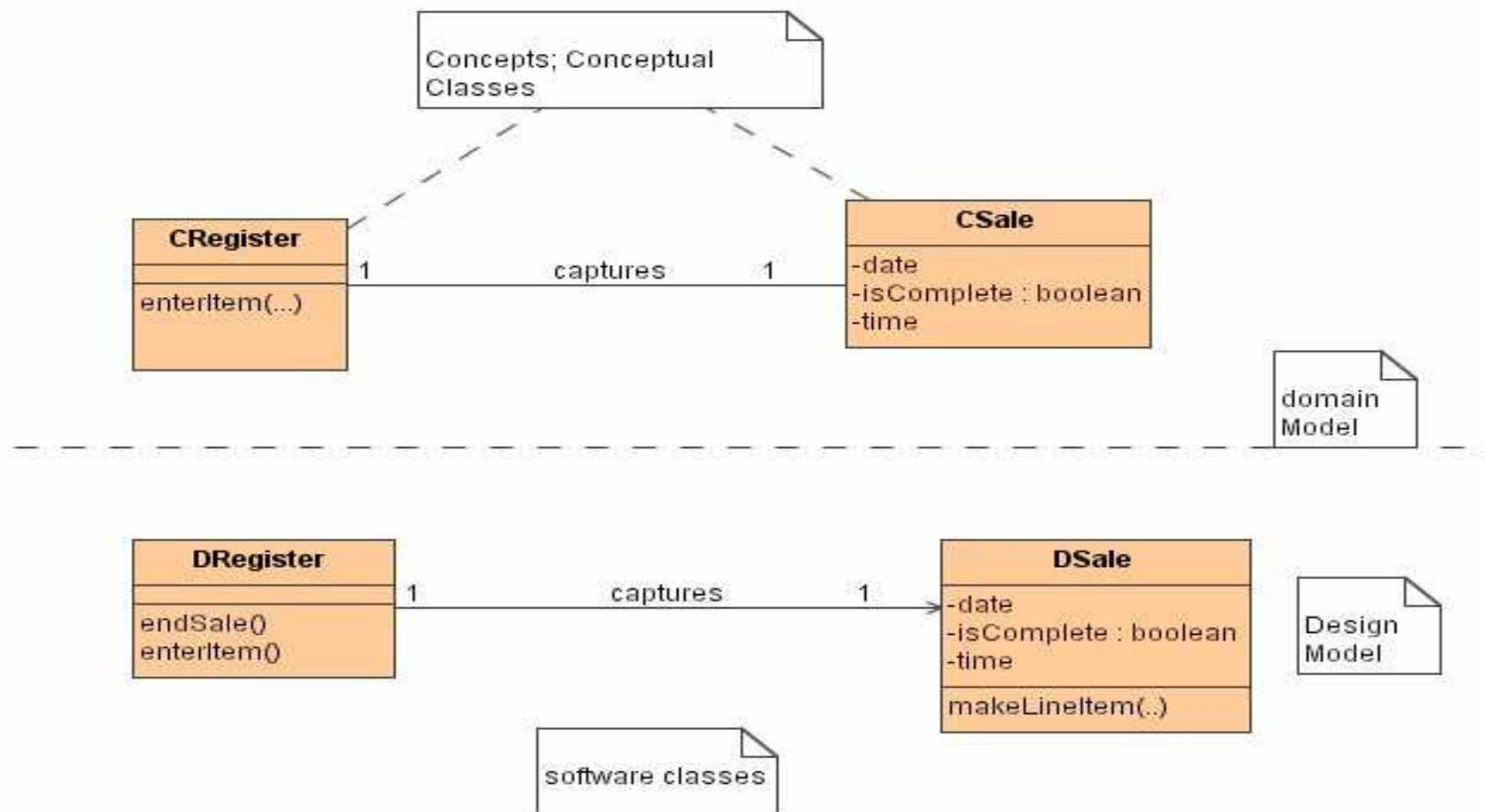
# Domain Vs. Design Classes

In a domain model, a class does not represent a software definition. Rather it is an abstraction of a real world-world concept about which we are interested in making statement.

Design Classes express the definitions of classes as software components. In these diagram, a class represent a software class.

In contrasts to conceptual classes in the domain model, design classes in the DCD show definitions for software classes rather than real-world concepts.

# Example: CCD and DCD



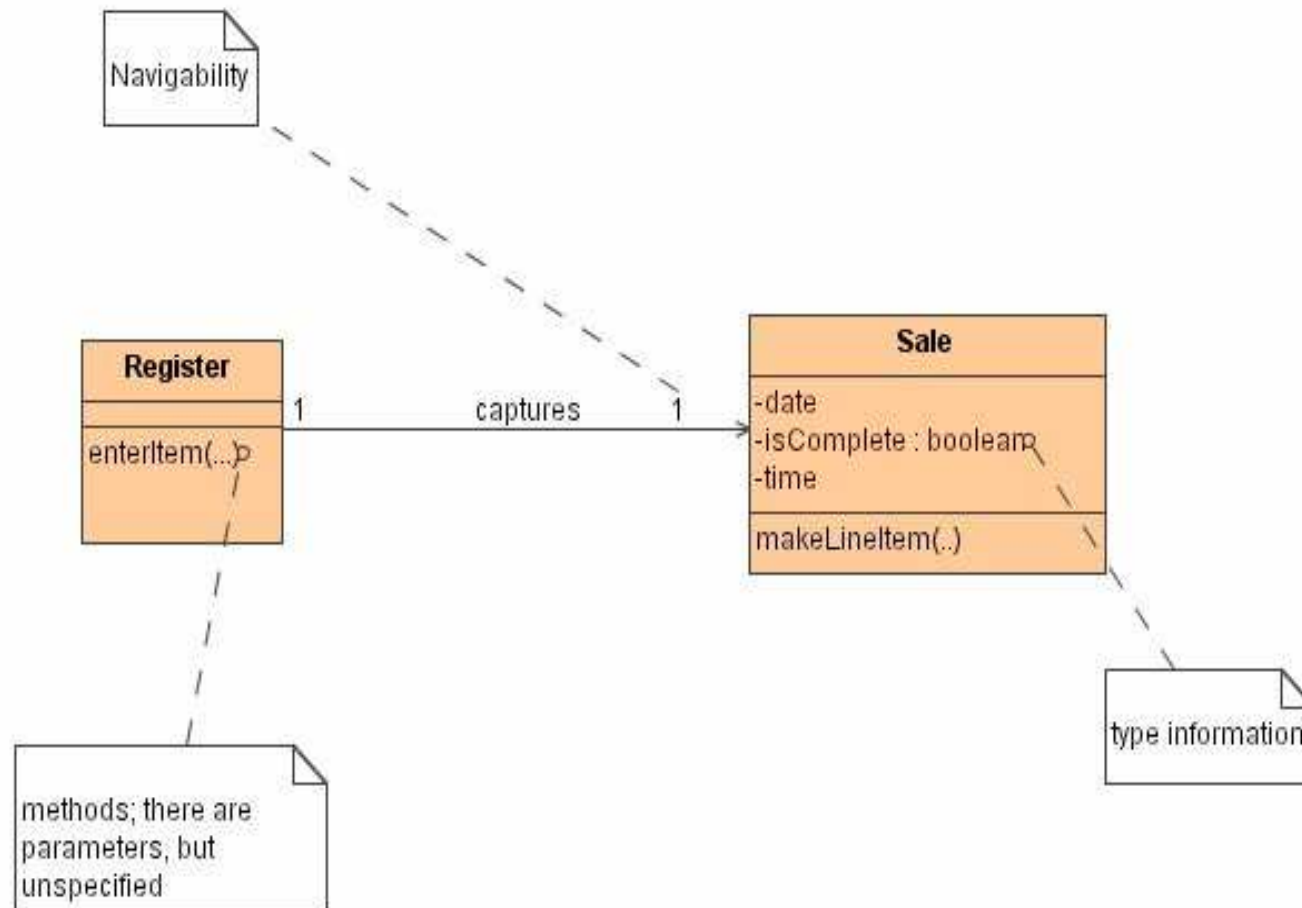
# Creating Design Class Diagrams 1

- 1) identify all classes participating in object interaction by analyzing the collaborations
- 2) present them in a class diagram
- 3) copy attributes from the associated concepts in the conceptual model
- 4) add methods names by analyzing the interaction diagrams
- 5) add type information to the attributes and methods

# Creating Design Class Diagrams 2

- 6) add the association necessary to support the required attribute visibility
- 7) add navigability arrows necessary to the associations to indicate the direction of the attribute visibility
- 8) add dependency relationship lines to indicate non-attribute visibility

# Example: DCD



# Adding Classes

---

The first step in creating a design class diagram is to identify those classes that participate in the software solution.

These classes can be found by scanning all the interaction diagrams and listing the participating classes.

# Adding Attributes

---

After defining the classes, attributes must be added.

Attributes come from software requirements artifacts, for example conceptual classes provide most of the attributes applicable to design classes.



# Adding Methods

---

The methods of each class can be identified by analyzing the interaction diagrams.

For example if the message *displayLogOnform* is sent to an instance of the *LicenseApplicationClient* class, then the class *LicenseApplicationClient* must define a method *displayLogOnform*.

# Adding Visibility

---

## Visibility:

- 1) scope of access allowed to a member of a class
- 2) applies to attributes and operations

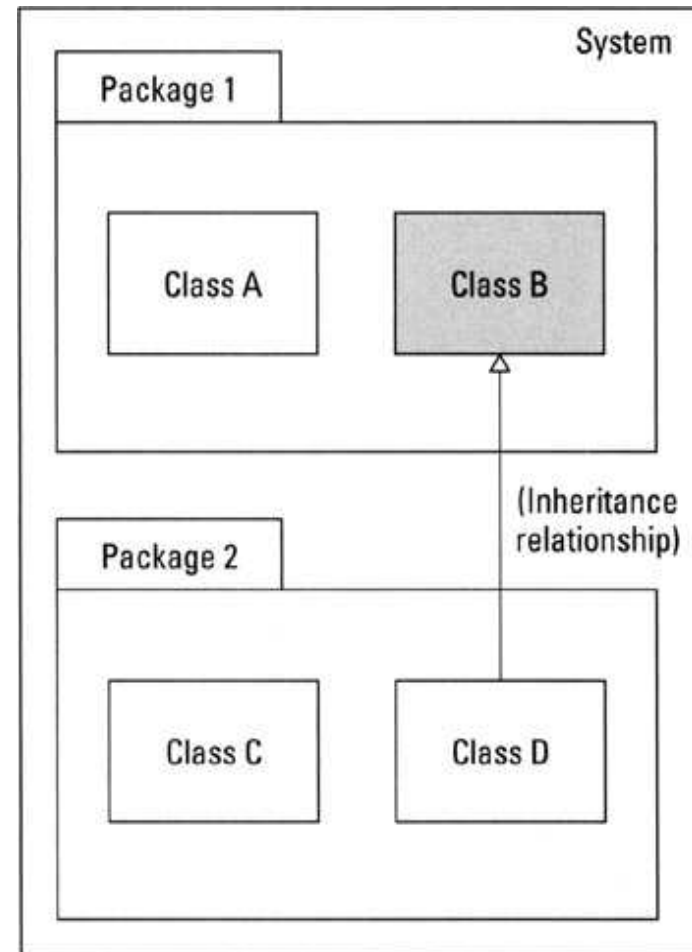
UML visibility maps to OO visibilities:

- 1) **Private scope**: within a class (-)
- 2) **package scope**: within a package (~)
- 3) **public scope**: within a system (+)
- 4) **protected scope**: within an inheritance tree (#)

# Private Visibility

---

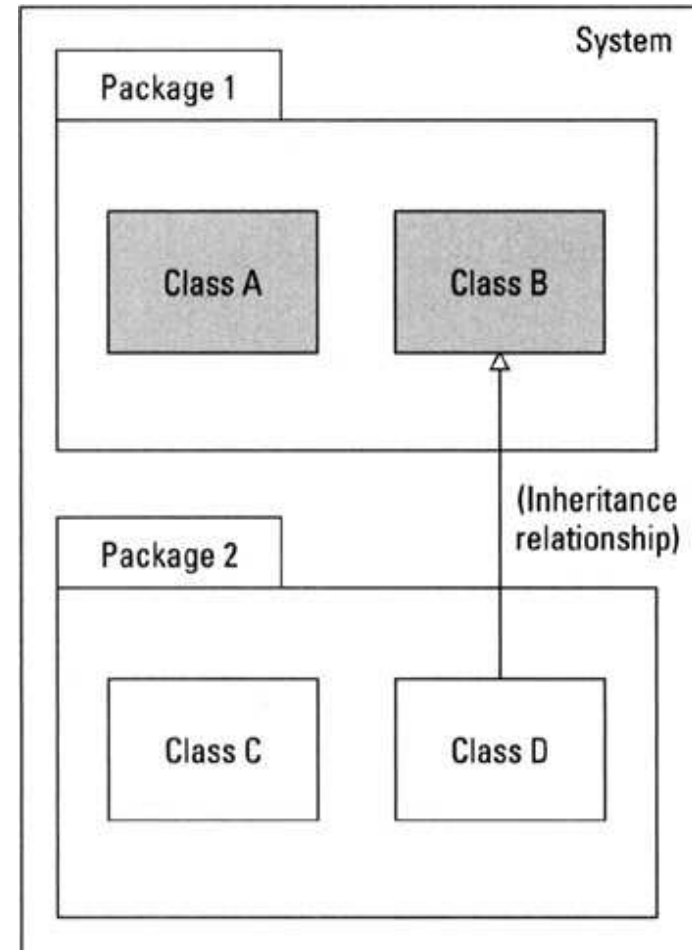
- 1) private element is only visible inside the namespace that owns it
- 2) notation is "-"
- 3) useful in encapsulation



# Package Visibility

1) package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace

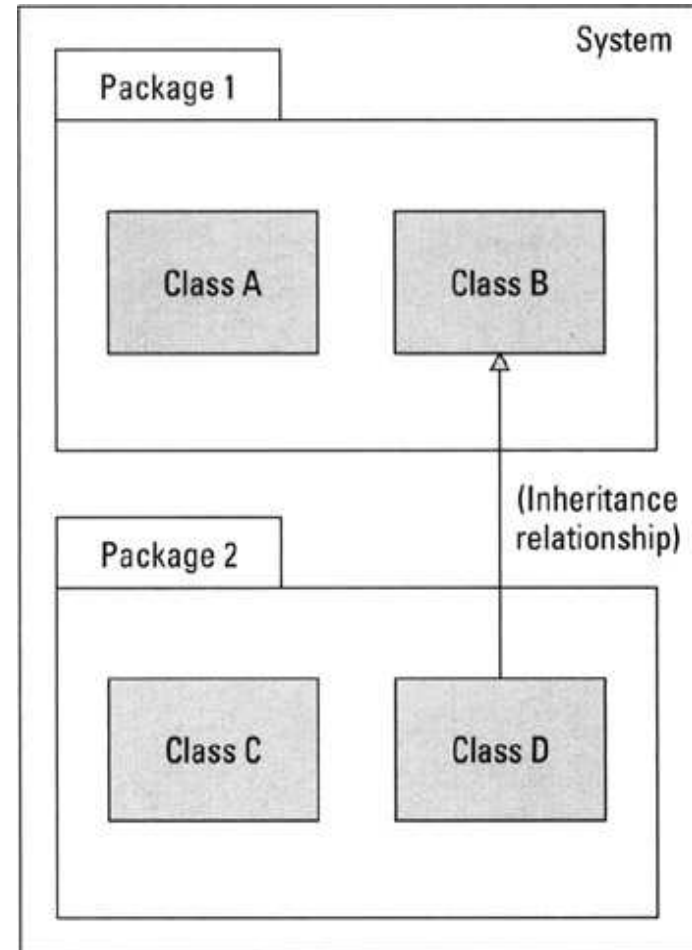
2) notation is "~"



# Public Visibility

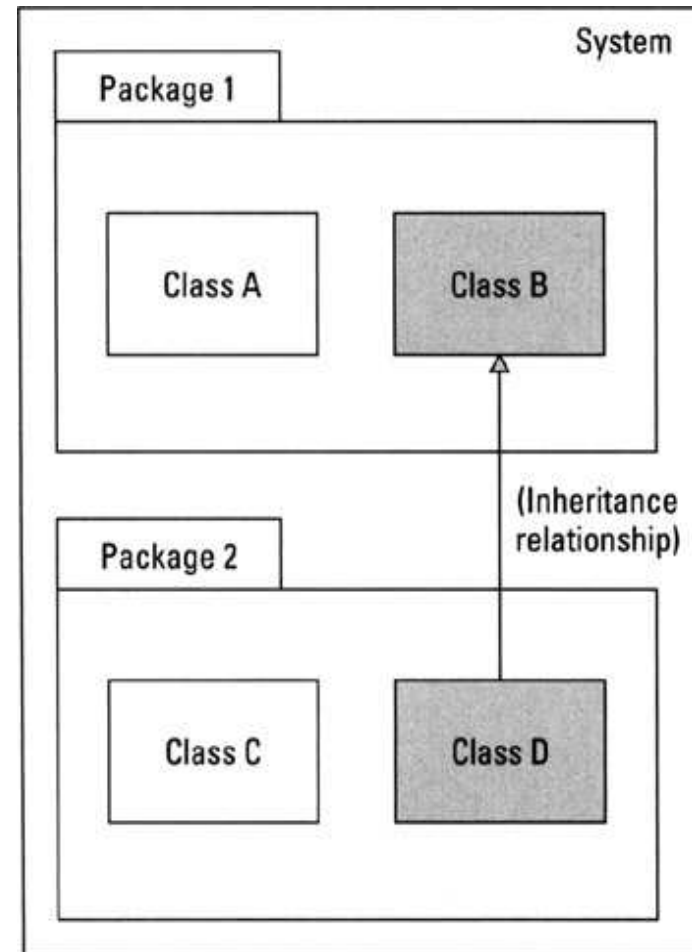
---

- 1) public element is visible to all elements that can access the contents of the namespace that owns it
- 2) notation is "+"



# Protected Visibility

- 1) a protected element is visible to elements that are in the generalization relationship to the namespace that owns it
- 2) notation is “#”



# Adding Associations

---

The end of an association is called a **role**.

In DCDs the role may be decorated with a navigability arrow. Navigability is a property of a role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target.

Navigability indicates attributes visibility.

During implementation in an object-oriented programming language it is usually translated as the source class having an attribute that refers to an instance of the target class.

Most if not all associations in a DCD should be adorned with the necessary navigability arrows.

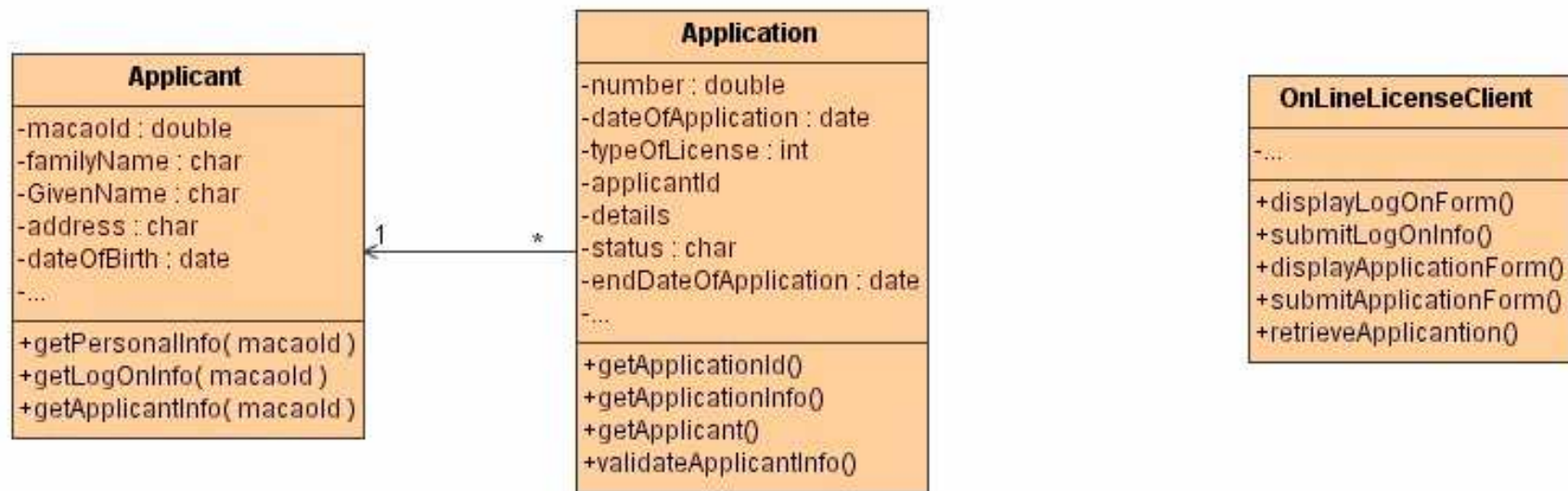
# Adding Dependency

In class diagrams, the dependency relationship is useful to depict non-attribute visibility between classes.

For example dependency relationships are useful when there are parameters, global or locally declared visibility.



# Example: Case Study



# Design Sequence Diagrams

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

# Design Sequence Diagrams

They present the interactions between objects needed to provide a specific behaviour.

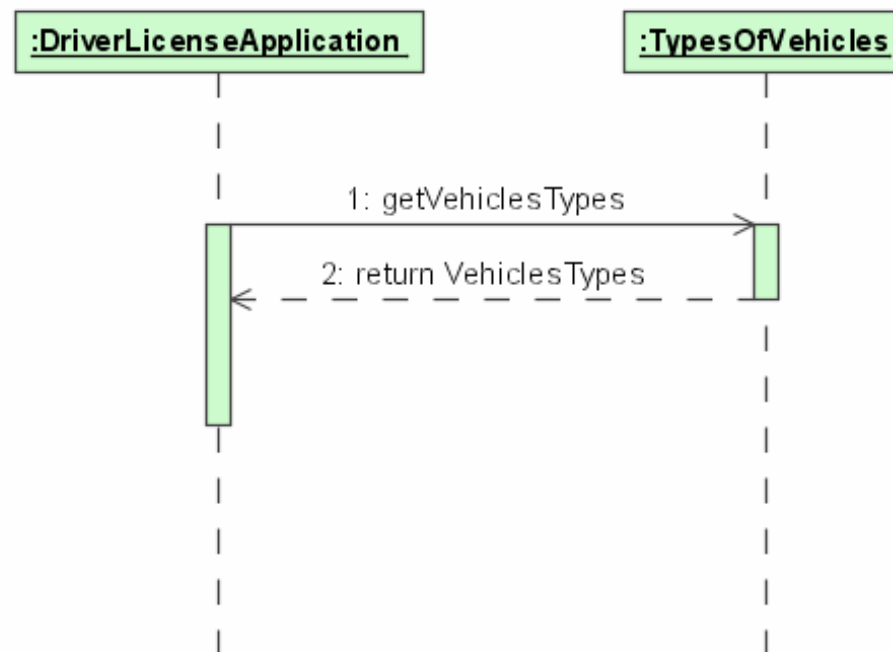
They capture the sequence of events between participating objects, taking into account temporal ordering and optionally which objects are active at any time.

Design considerations:


- a) different types of message
- b) timed-messages
- c) activation and deactivation
- d) recursion
- e) object creation and destruction

# Synchronous Messages

**Synchronous message:** assumes that a return is needed, so the sender waits for the return before proceeding with any other activity.

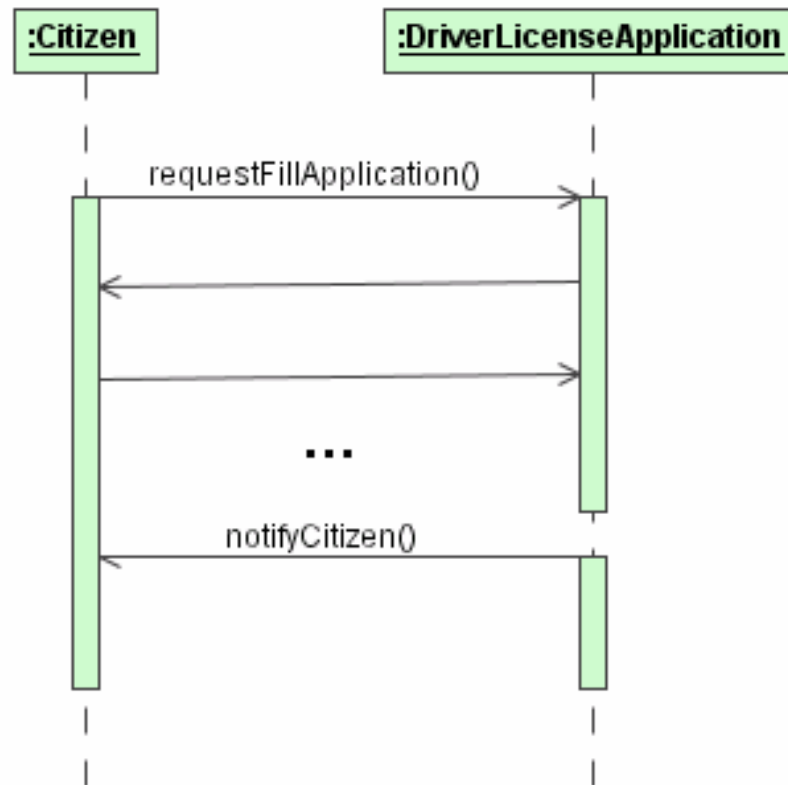


# Asynchronous Messages

- 1) **asynchronous message**: there is no need for an answer
- 2) signals are asynchronous messages
- 3) the sender is responsible only to get the message to the receiver and it has no responsibility or obligation to wait
- 4) the receiver may decide to do nothing or to process the received message
- 5) an asynchronous message is modelled using a solid line and a half arrowhead to distinguish it from the full arrowhead of the synchronous message
- 6) notation 

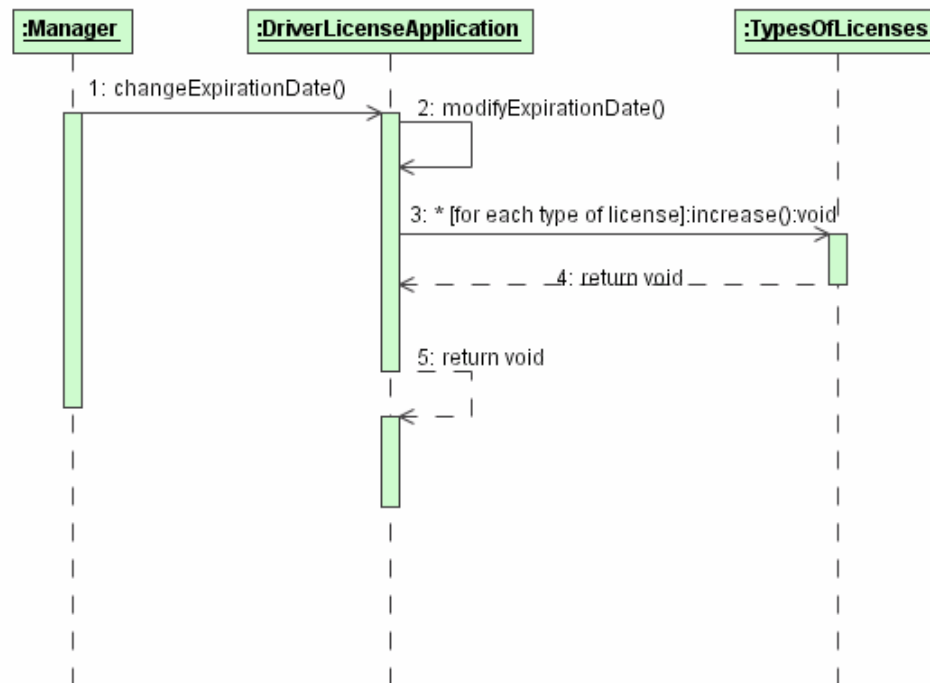
# Example: Messages

---



# Self-Reference Message

A **self-reference message** is a message where the sender and receiver are one and the same object.



- 1) in a self-reference message the object refers to itself when it makes the call
- 2) message 2 is only the invocation of some procedure that should be executed



# Timed Messages

---

A message may have any number of user-defined time attributes, such as **sentTime** or **receivedTime**.

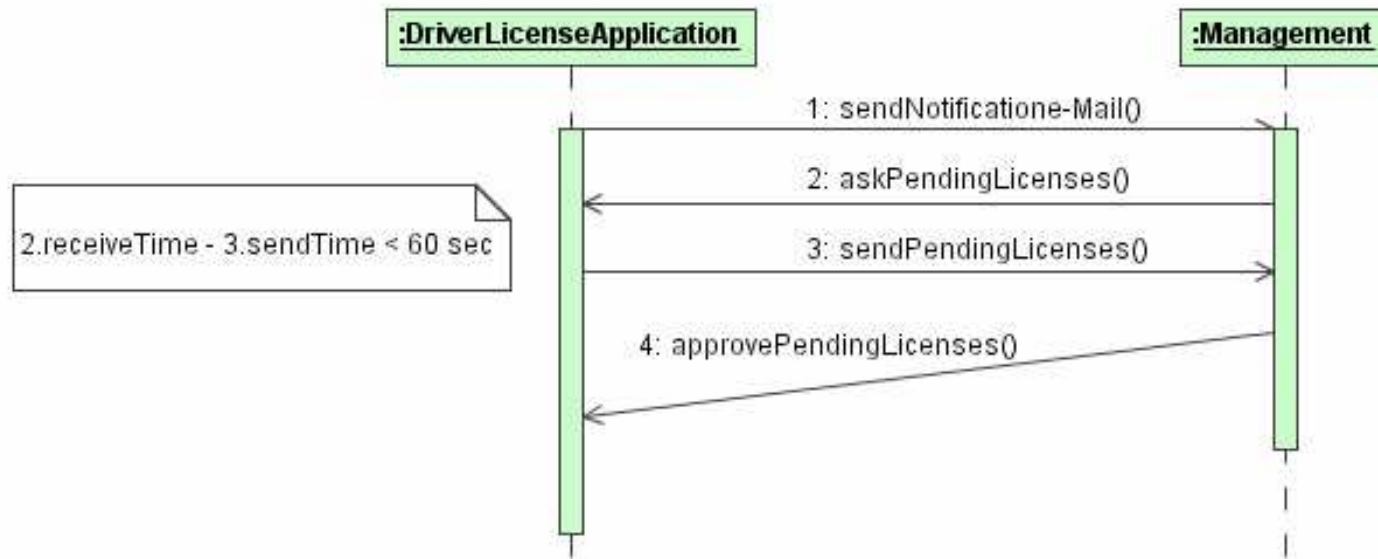
To access these values you **need to identify the message** that owns the value by using the message number or the message name.

This identification is useful to specify **time constraints** that may be described at the left margin.

Instantaneous messages are modelled with horizontal arrows.

For messages requiring a significant amount of time, it is possible to slant the arrow from the tail down to the head.

# Example: Timed Messages



For messages 1, 2 and 3 the time required for their execution is considered equal to zero.

Message 4 requires more time ( $time > 0$ ) for its execution.

# Activation and Deactivation

Sequence diagrams can show object activation and deactivation.

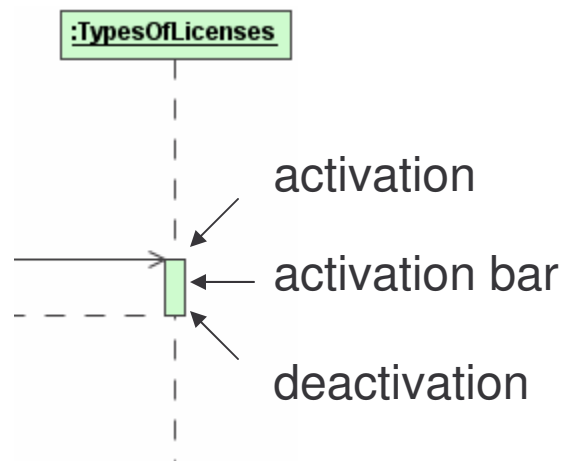
**Activation** means that an object is occupied performing a task.

**Deactivation** means that the object is idle, waiting for a message.

# Activation and Focus of Control

Activation is shown by widening the vertical object lifeline to a narrow rectangle, called an **activation bar** or **focus of control**.

An object becomes active at the top of the rectangle and is deactivated when control reaches the bottom of the rectangle.



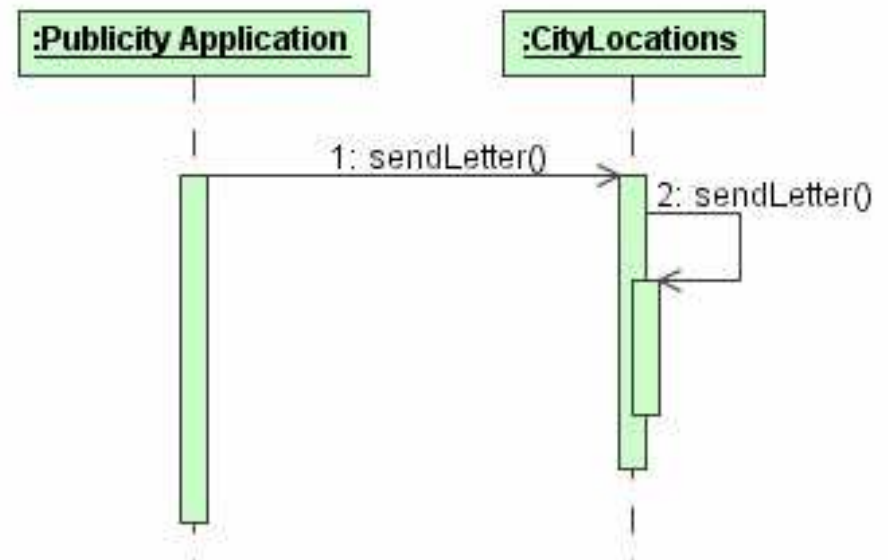
# Recursion

---

An object might also need to call a message recursively, this means to call the same message from within the message.

Suppose that cityLocations is defined in the class diagram as a set of one or more apartments or houses.

You need to send a letter to all locations, so if a location is compound, the letter should be sent to all components.



# Creation and Destruction

---

## Object Creation:

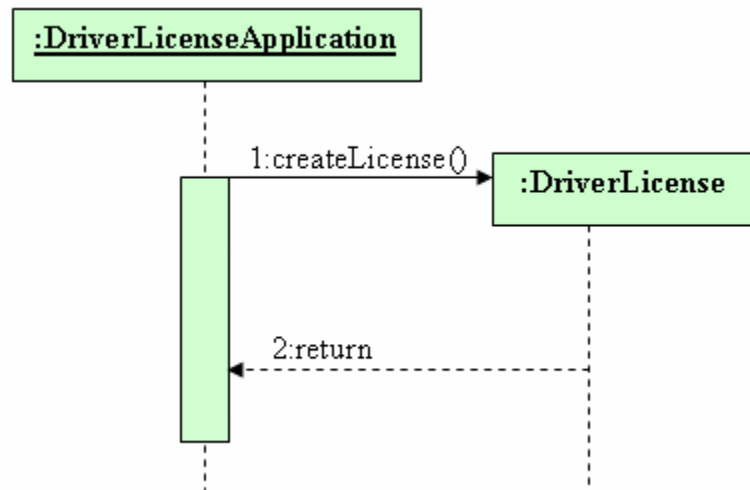
- if the object is created during the sequence execution it should appear somewhere below the top of the diagram.

## Object Destruction:

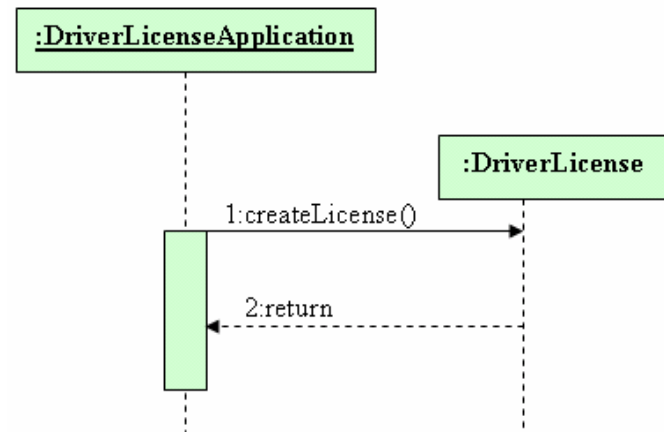
- if the object is deleted during the sequence execution, place an X at the point in the object lifeline when the termination occurs.

# Examples: Object Creation

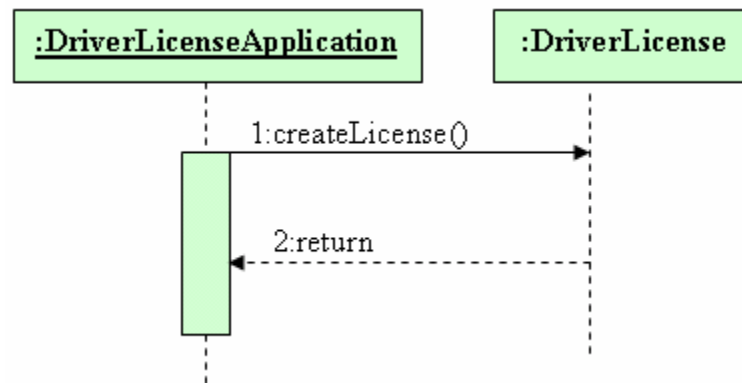
Alternative A



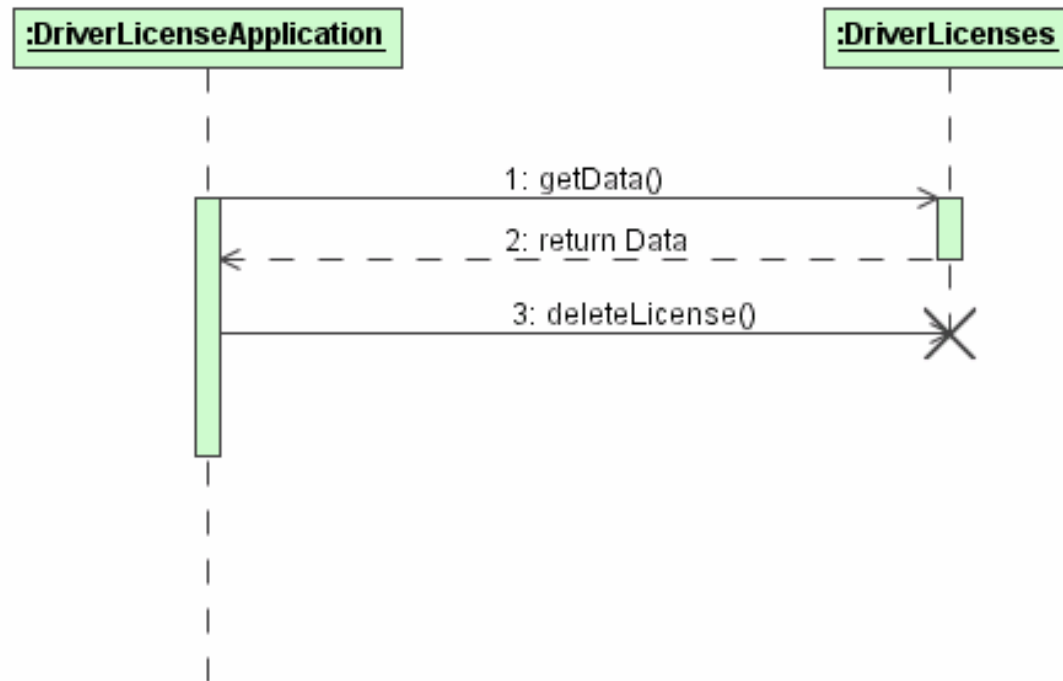
Alternative B



Alternative C



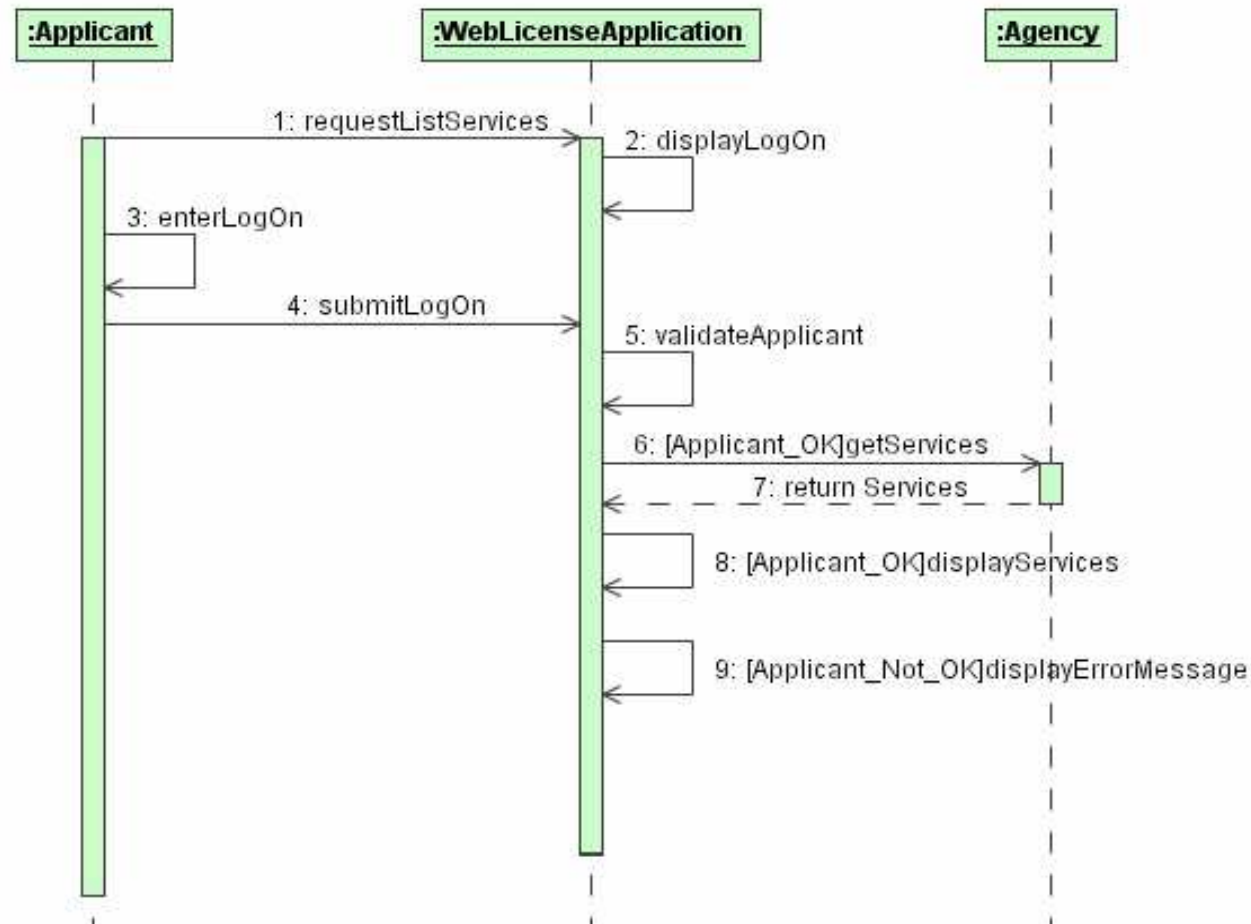
# Example: Object Destruction



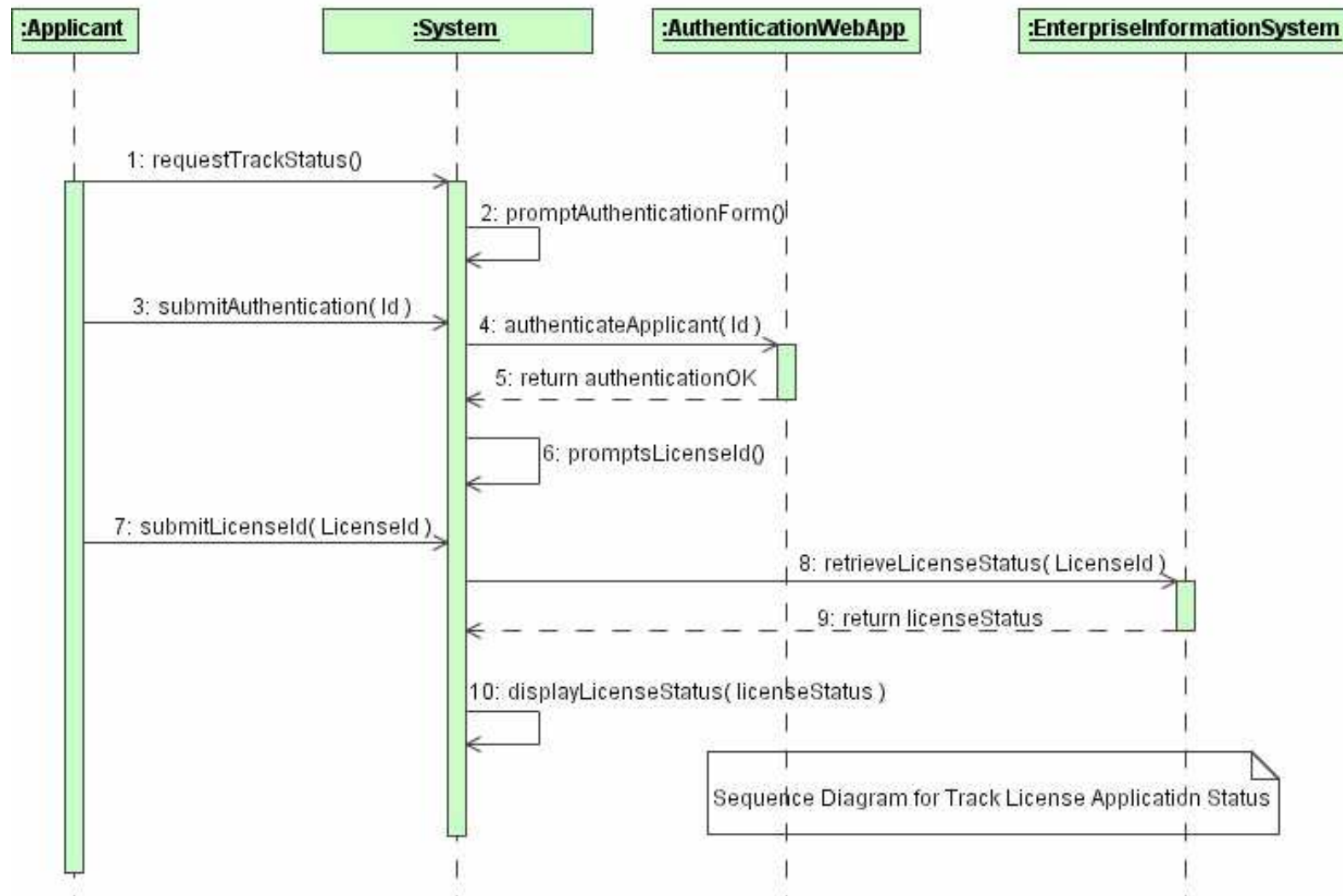


# Advanced Features

It is possible to combine several scenarios in one sequence diagram:



# Example: Case Study



# Activity Diagrams

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

# Activity Diagrams 1

- 1) it is a variation of the state machine in which the states represent the performance of actions or sub-activities and the transitions are triggered by the completion of the actions or sub-activities
- 2) one of the five diagrams in UML for modelling the dynamic aspect of a system.  
Others: **sequence**, **collaboration**, **statechart** and **use cases**
- 3) it is attached through a model to a classifier, such as a use case, or to a package, or to the implementation of an operation
- 4) its purpose is to focus on flows driven by internal processing (as opposed to external events)

# Activity Diagrams 2

- 5) it involves modelling the **dynamic aspects** of a system; usually the sequential (but possibly concurrent) steps in a computational process
- 6) activity diagrams also model the **flow of an object** as it moves from one state to another at different points in the flow of control
- 7) they are used in situations where all or most of the events represent the completion of **internally-generated actions** (procedural flow of controls)

# Action State

---

An action state is a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action.

There may be several outgoing transitions with guard conditions.

Action states may not have:

- a) internal transitions
- b) outgoing transitions based on explicit events or exit actions

It models a step in the execution of a workflow.

# Action State – Notation

---

It is shown as a shape with straight top and bottom and with convex arcs on the two sides.

Action expression is placed in the action state symbol.

Action expression may be described as natural language, pseudocode, action language or programming language.





# Subactivity State

---

It invokes an activity graph.

When a subactivity state is entered, the activity graph nested in it is executed as any activity graph would be.

The subactivity graph is not exited until the final state of the nested graph is reached.

A single activity graph may be invoked by many subactivity states.

# Subactivity State – Notation

It is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram.

The name of the subactivity is placed in the symbol.



# Transitions

---

They specify the flow of control from one action or activity state to another.

Transitions in activity diagrams are triggerless.

Control passes immediately once the work of the source state is done.

Once the action of a given source state completes, the state's exit action (if any) is executed; next without delay control follows the transition and passes on to the next action or activity state.

# Decisions

---

Decisions are expressed as guard conditions.

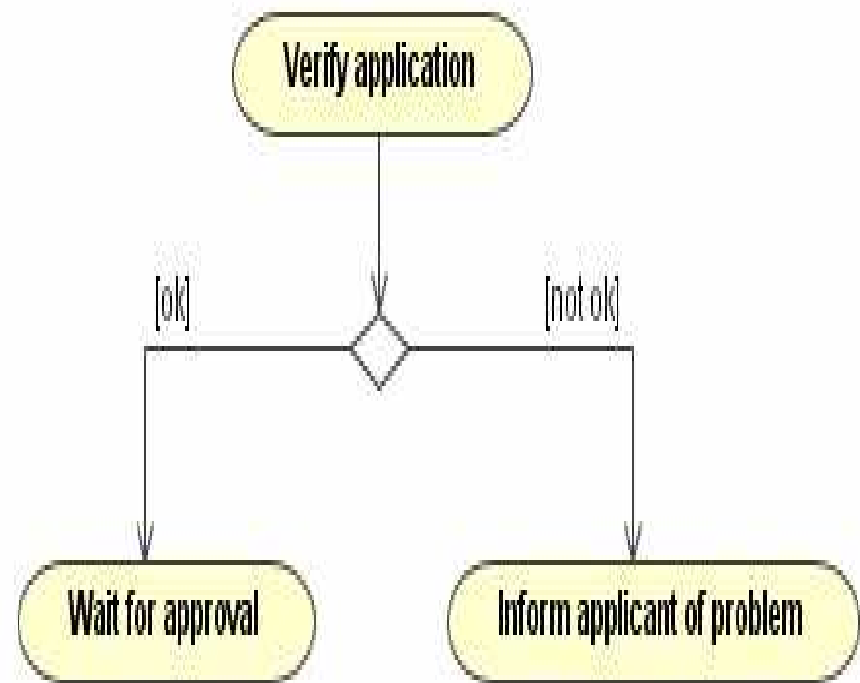
Different guards are used to indicate different possible transitions that depend on boolean conditions of the owning objects.

The predefined guard `else` may be defined for at most one outgoing transition which is enabled if all the guards labeling the other transitions are false.

# Decision - Notation

Decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each with a distinct guard condition with no event trigger.



# Forks and Joins 1

---

They are used for modelling concurrency and synchronization in business processes.

Synchronization bar is used to specify the forking and joining of these parallel flows of control.

A fork represents the splitting of a single flow of control into two or more concurrent flows of control.

Activities of each of these flows are truly concurrent (in the case of a system deployed across multiple nodes) or sequentially interleaved if deployed on a single node.

# Forks and Joins 2

---

A join represents the synchronization of two or more concurrent flows of control.

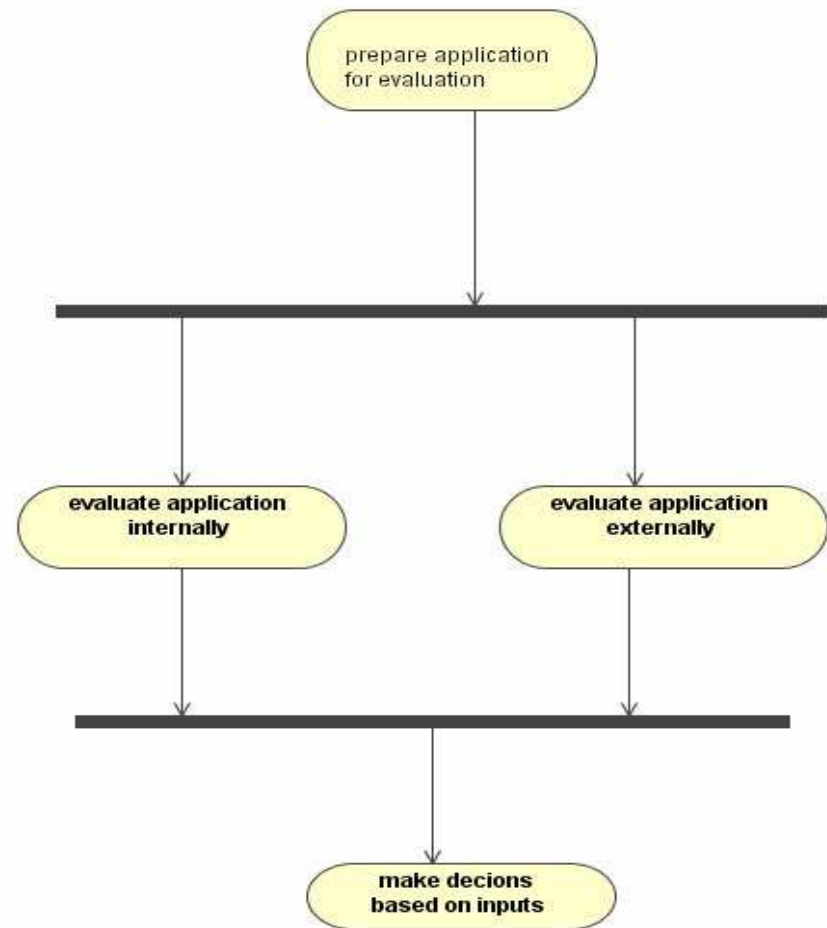
Joins may have two or more incoming transitions and one outgoing transition.

Above a join, the activities associated with each of these paths continues in parallel.

At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join at which point one flow of control continues on below the join.

# Forks and Join Example

- 1) application is prepared for evaluation
- 2) internal and external evaluation proceeds concurrently
- 3) decision is made only when the internal and external evaluation are completed



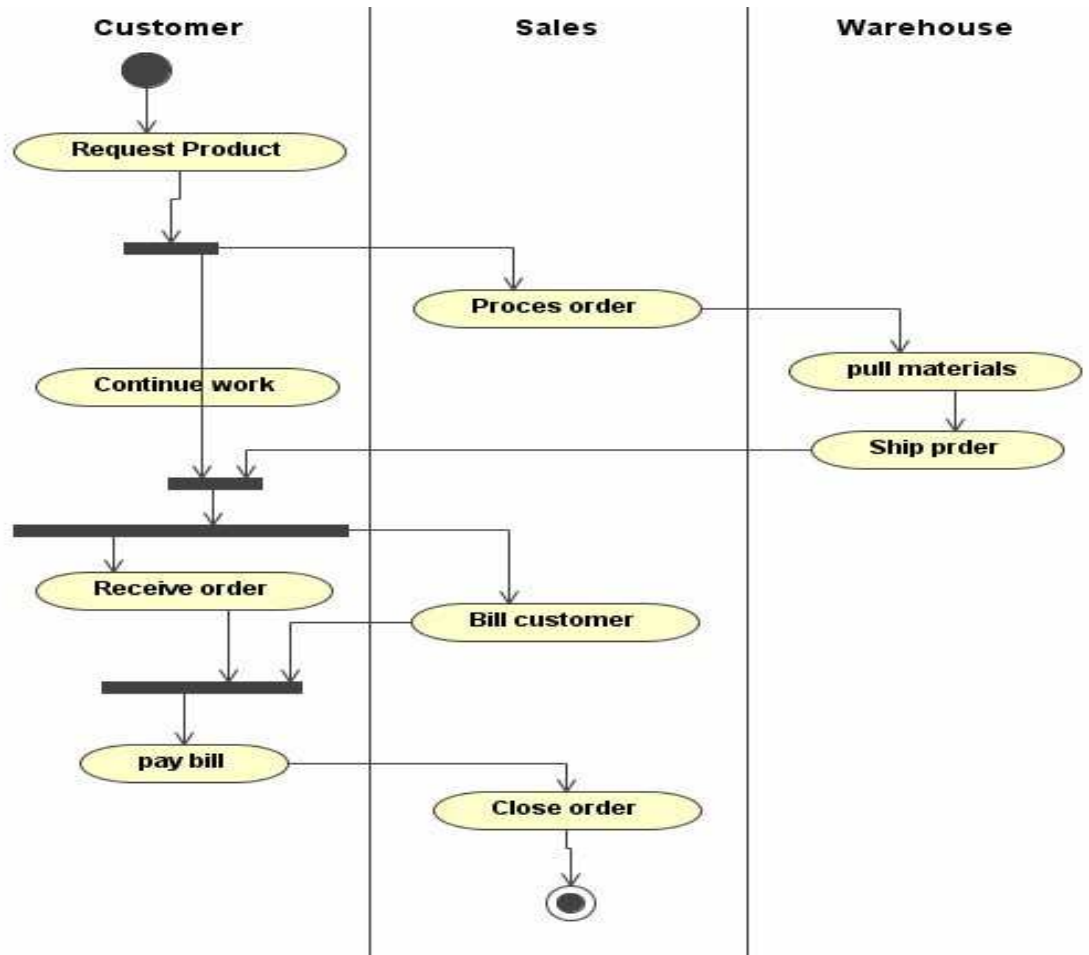


# Swimlanes

---

- 1) we may wish to partition the activity states on an activity diagram into groups
- 2) these groups may represent the business organization responsible for those activities
- 3) each of these groups is called a swimlane and it specifies a locus of activities.
- 4) each swimlane has a unique name within a diagram

# Example: Swimlanes

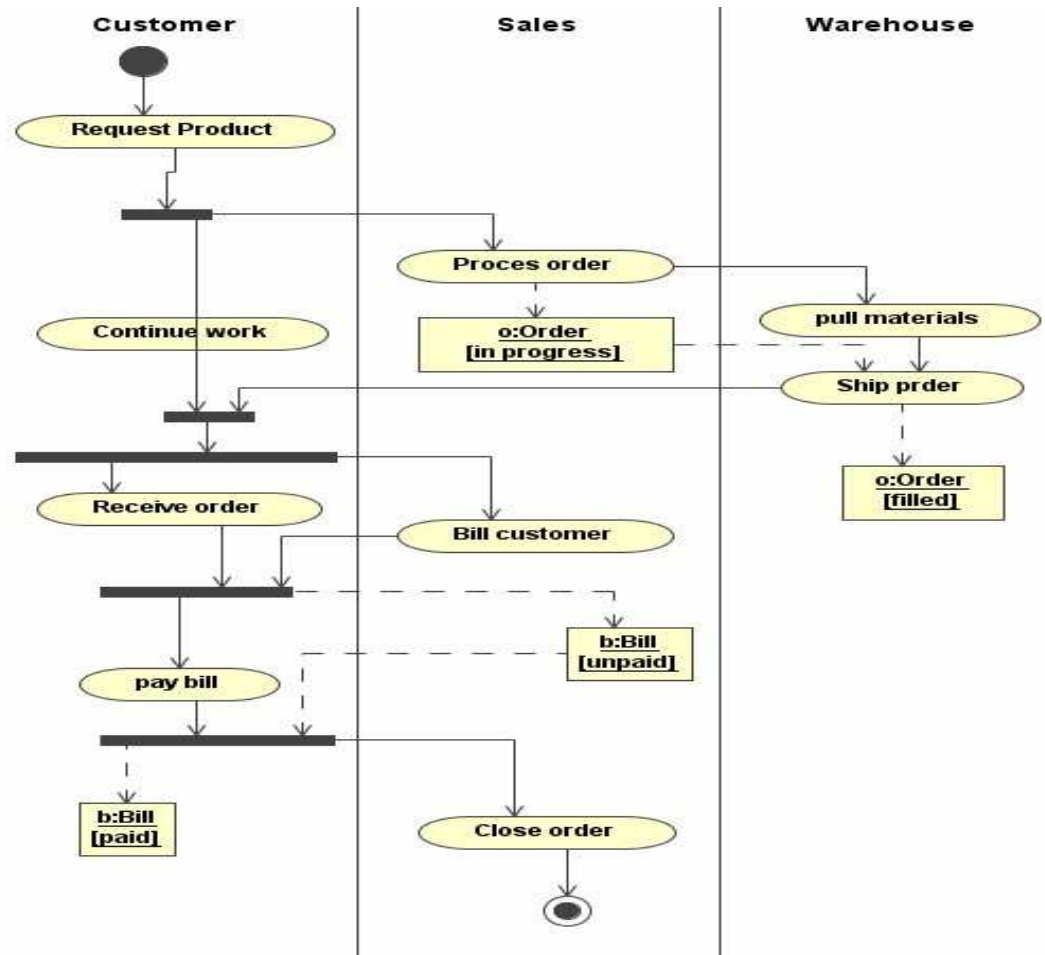


# Object Flow

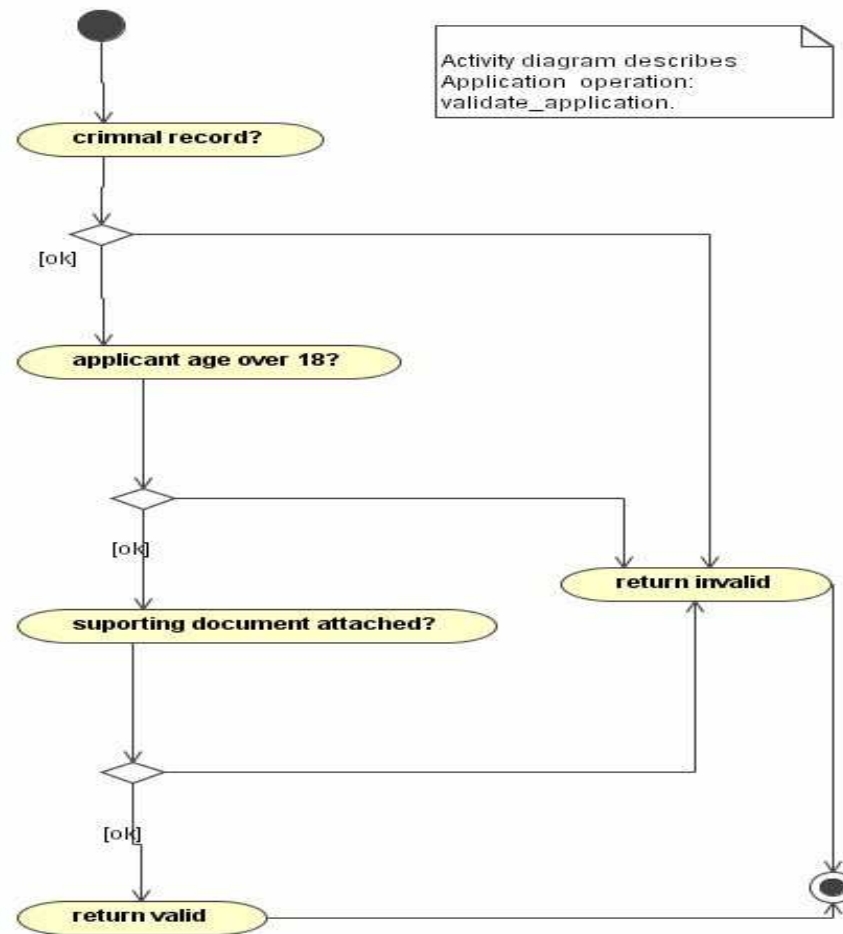
---

- 1) objects may be involved in the flow of control associated with an activity diagram
- 2) for example: in the workflow of processing an order as shown in the last example, we may wish to show how the state of the *Order* and *Bill* object changes.
- 3) we can use dependency relationship to show how objects are created, modified and destroyed by transitions in the activity diagram

# Example: Object Flow



# Example: Case Study



# Design Statechart Diagrams

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

# Design Statechart Diagrams

Collaboration and sequence diagrams are used to understand how the objects collaborate within the system.

Statechart diagrams illustrate how these objects behave internally.

Statecharts diagrams relate events to state transitions and states.

The transitions change the state of the system and are triggered by events.

During the design phase we add more implementation details to the statechart diagrams or we develop new statechart diagrams.



# Static Branch Point

---

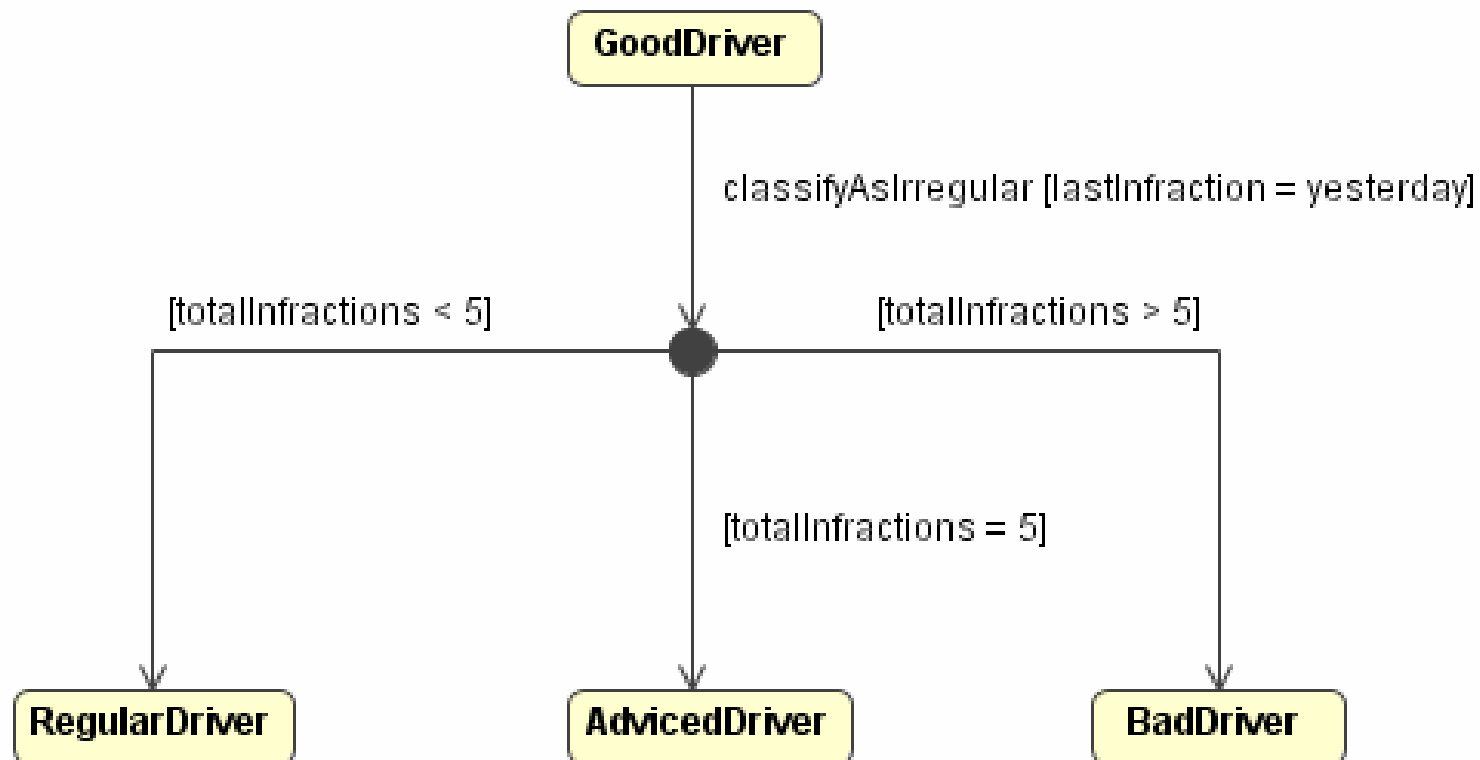
Another pseudo state provided by UML is the **static branch point** that allows a transition to split into two or more paths.

Notation:



It provides a means to simplify compound guard conditions by combining the like portions into a single transition segment, then branching based on the portions of the guard that are unique.

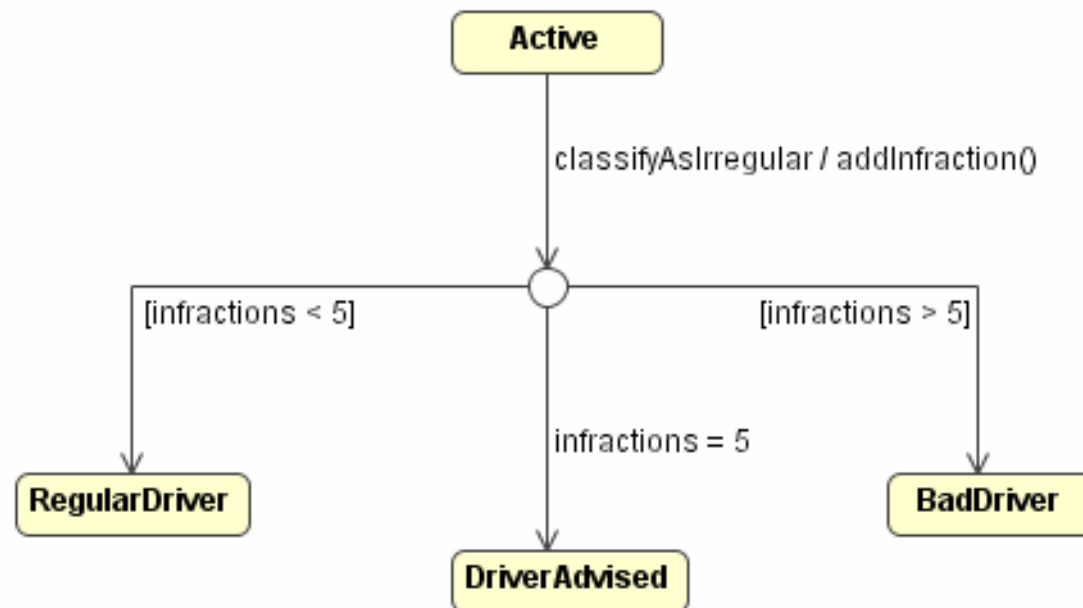
# Example: Static Branch Point



# Dynamic Branch Point

In other transitions, the destination is not known until the associated action has been completed.

Notation: ○



# Modelling Composite States

A **composite state** is simply a state that contains one or more statechart diagrams.

Composite states may contain either:

- 1) a set of mutually exclusive states: is literally like embedding a statechart diagram inside a state
- 2) a set of concurrent states: different states divided into regions, active at the same time

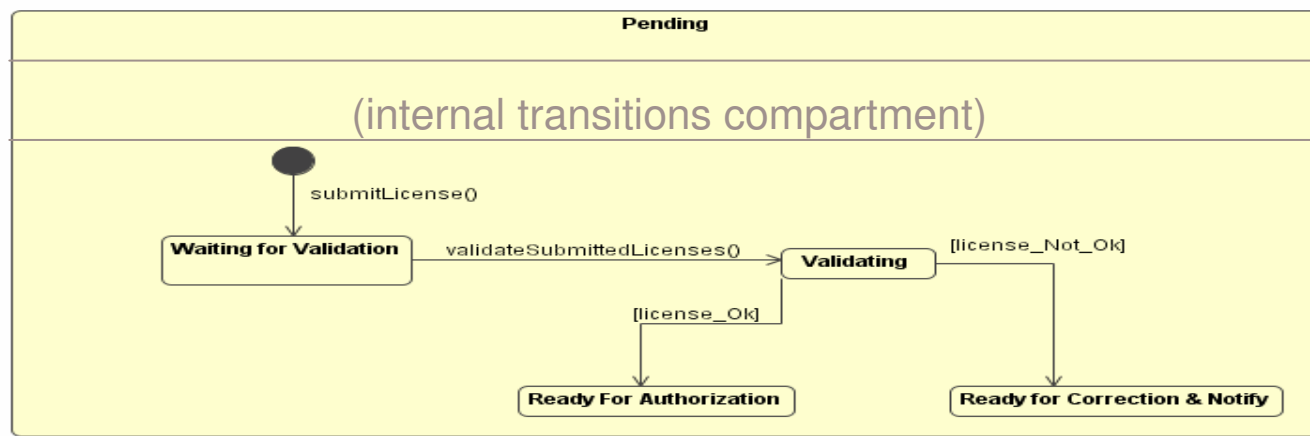
A composite state is also called a **super-state**, a generalized state that contains a set of specialized states called **sub-states**.

# Sub-States

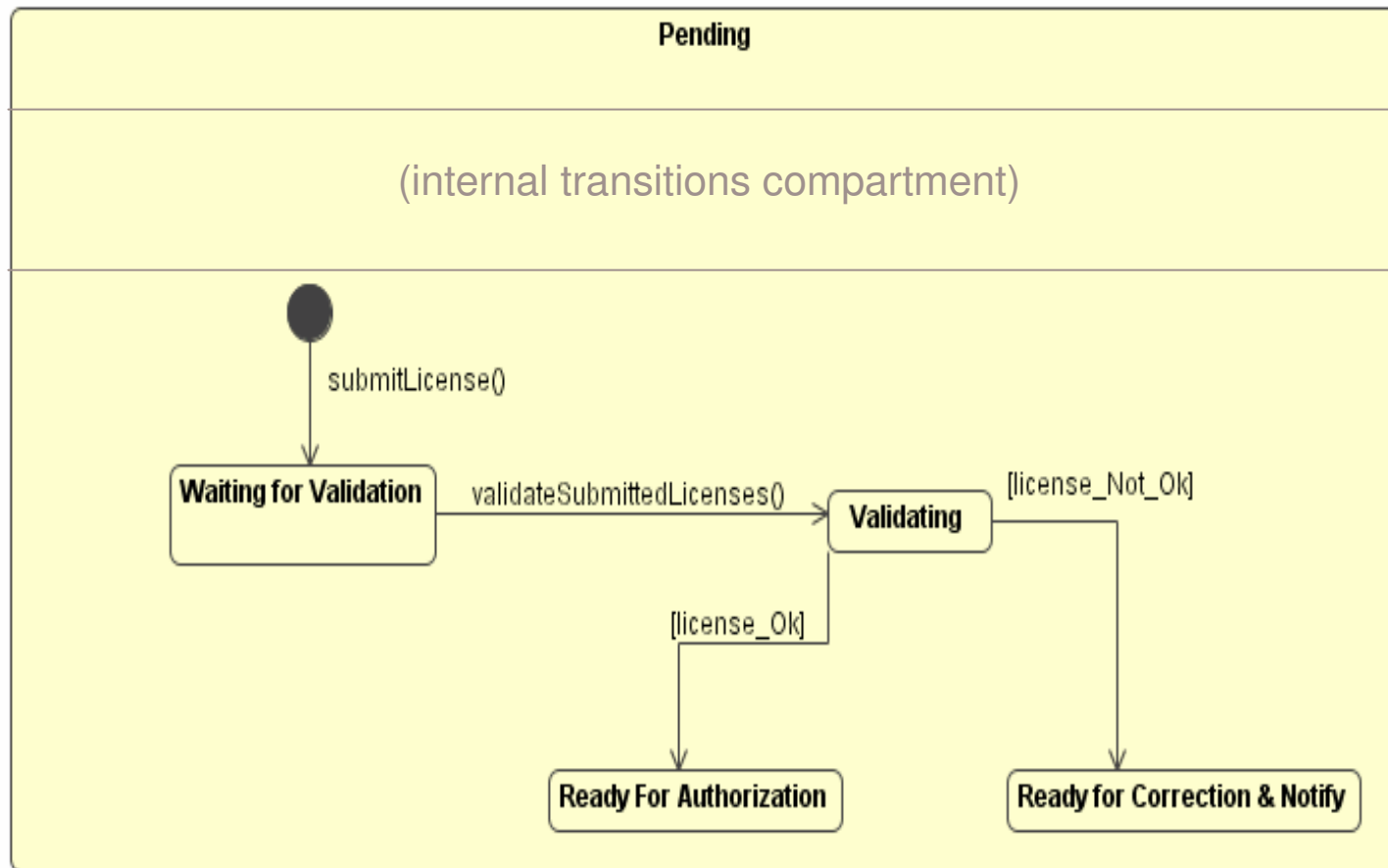
A composite state (super-state) may be decomposed into two or more lower-level states (sub-states).

All the rules and notation are the same for the contained sub-states as for any statechart diagram.

Decomposition may have as many levels as needed.



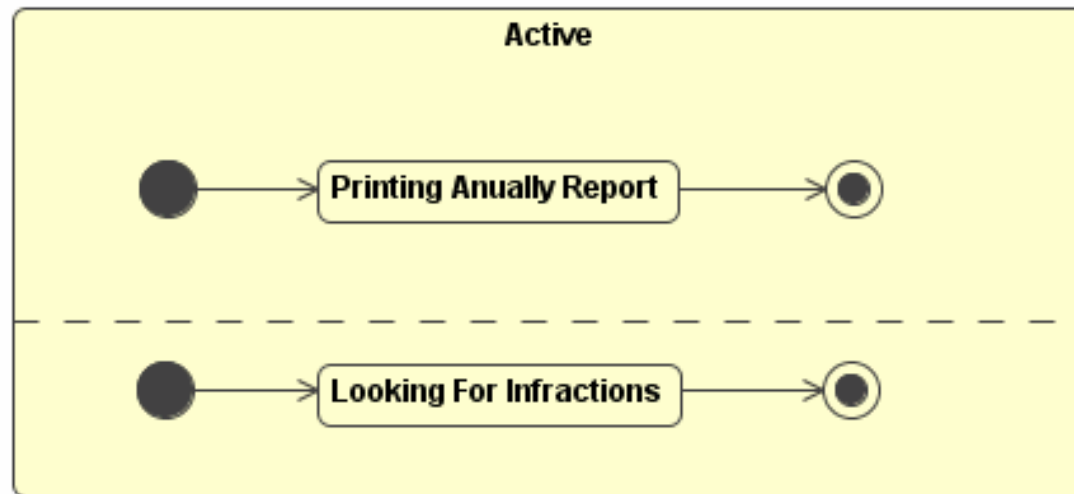
# Example: Sub-States



# Concurrent Sub-States

Modelling concurrent sub-states implies that you have many things occurring at the same time.

To isolate them, the composite state is divided into regions, and each region contains a distinct statechart diagram.



# Sub-Machine States

---

A **sub-machine state** is a kind of shorthand for referring to an existing statechart diagram.

Within a composite state, it is possible to reference to a sub-machine state in the same way that a class may call a subroutine or a function of another class.

The composite state containing the sub-machine is called the **containing state machine**, and the sub-machine is called the **referenced state machine**.

Access to sub-machines states is through entry and exit points that are specified by **stub states**.



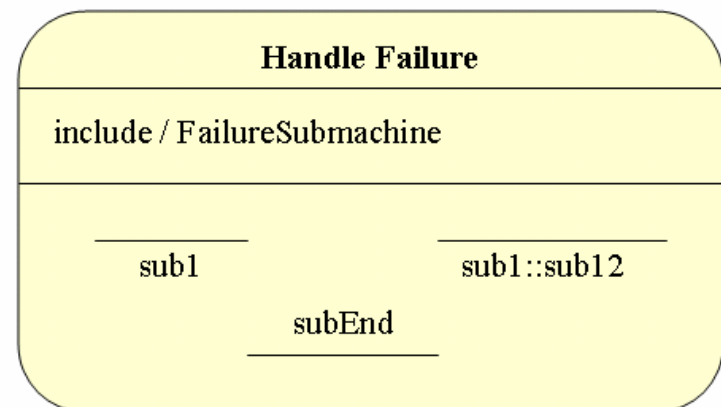
# Notation for Sub-Machine

The containing state icon models the reference to the sub-machine adding the keyword **include** followed by a slash plus the name of the submachine state.

The **stub state** uses a line with the state name placed near the line.

**Entry points** are represented with a line and the name under the line.

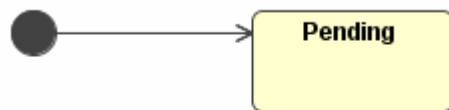
**Exit points** are represented with a line and the name over the line.



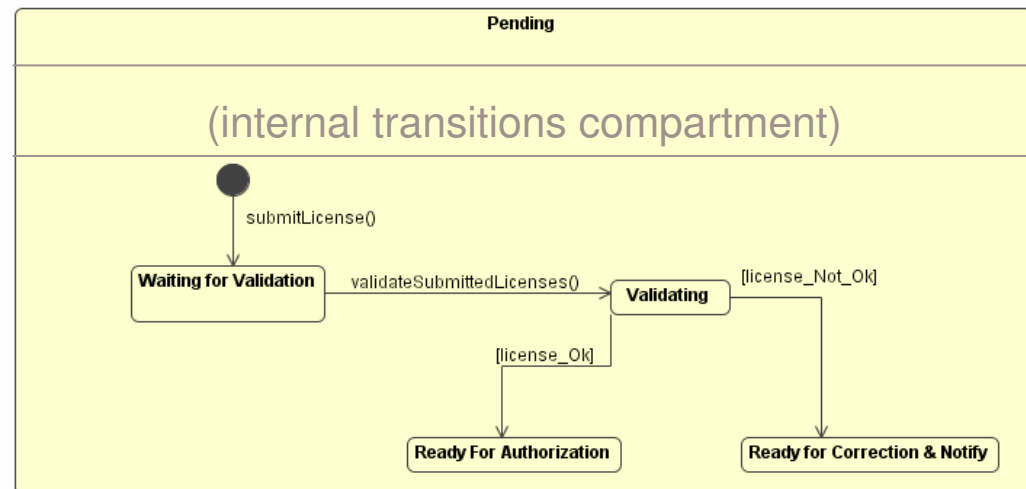
# Transitions to Sub-States 1

Alternative A:

- draw a transition pointing to the edge of the composite state icon
- it means that the composite state starts in the default initial state
- the default initial state is the sub-state associated with the initial state icon in the contained statechart diagram



The initial sub-state is **Waiting for Validation**

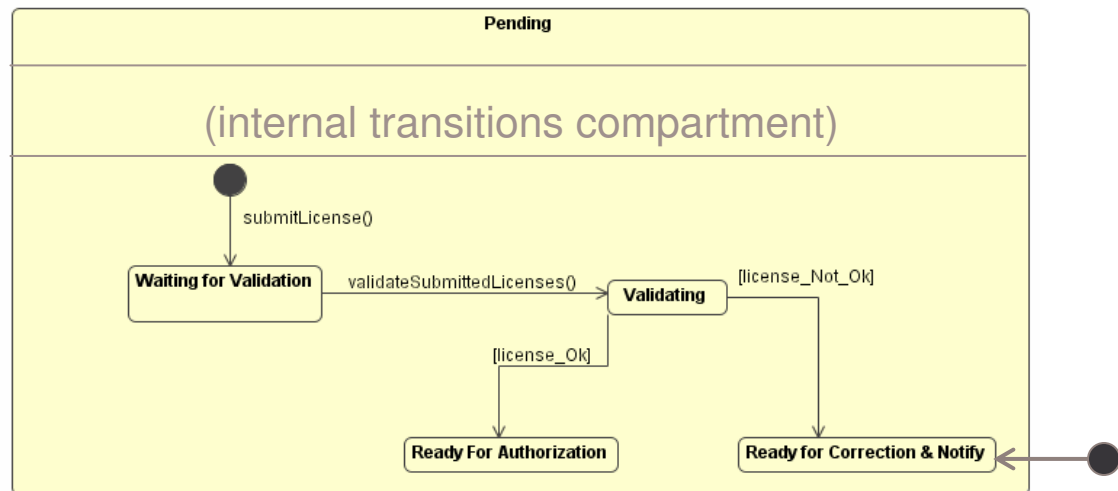


# Transitions to Sub-States 2

Alternative B:

- draw a transition through the edge of the super-state to the edge of the specific sub-state
- it means that the composite state starts in that specific sub-state

The initial sub-state  
is **Ready for  
Correction & Notify**

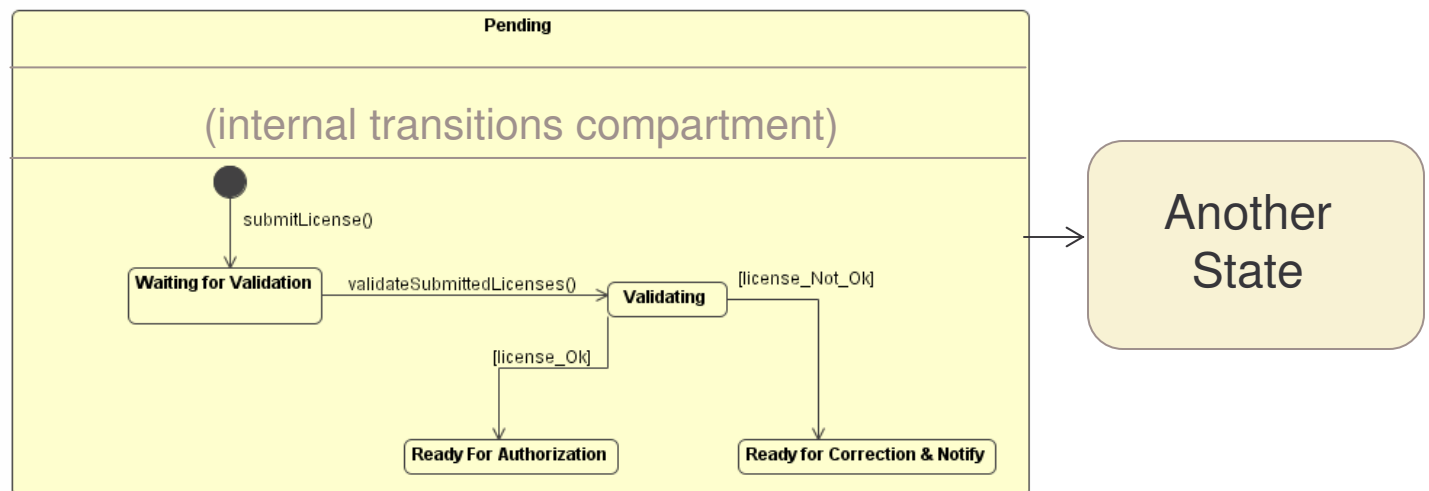


# Transitions from Sub-States 1

Alternative A:

- 1) an event can cause the object to leave the super-state regardless of the current sub-state
- 2) draw the transition from the edge of the super-state to the new state.

At any sub-state the object can do a transition to **AnotherState**

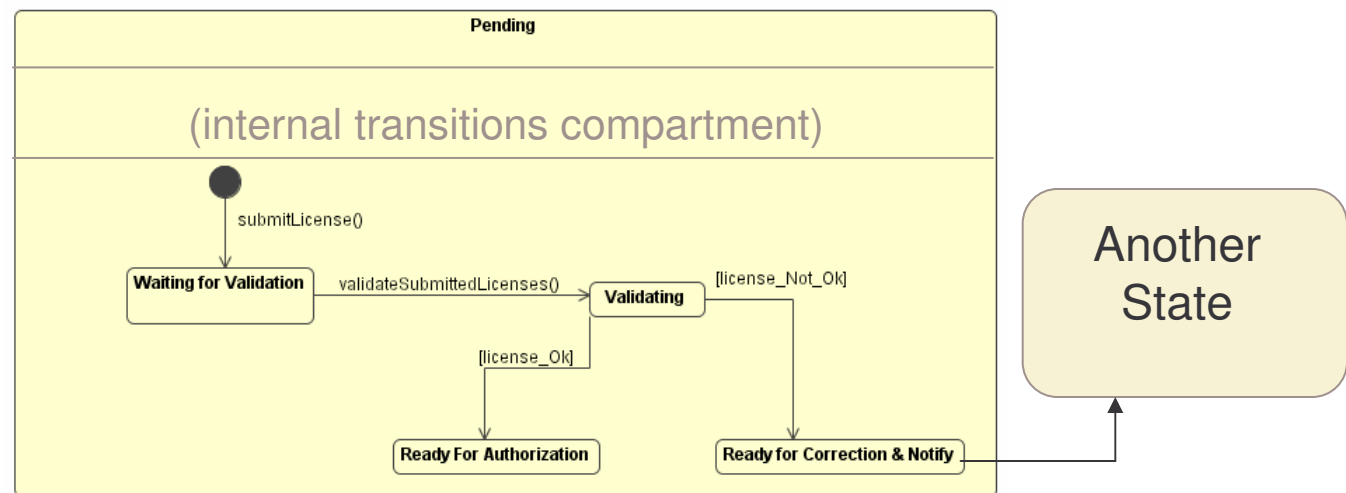


# Transitions from Sub-States 2

Alternative B:

- 1) an event can cause the object to leave the super-state directly from a specific sub-state
- 2) implies that the exit event may only happen when the object is in the specific sub-state
- 3) draw the transition from the edge of the specific sub-state through the edge of the super-state.

Only from sub-state **Ready for Correction & Notify** the object can make a transition to **AnotherState**

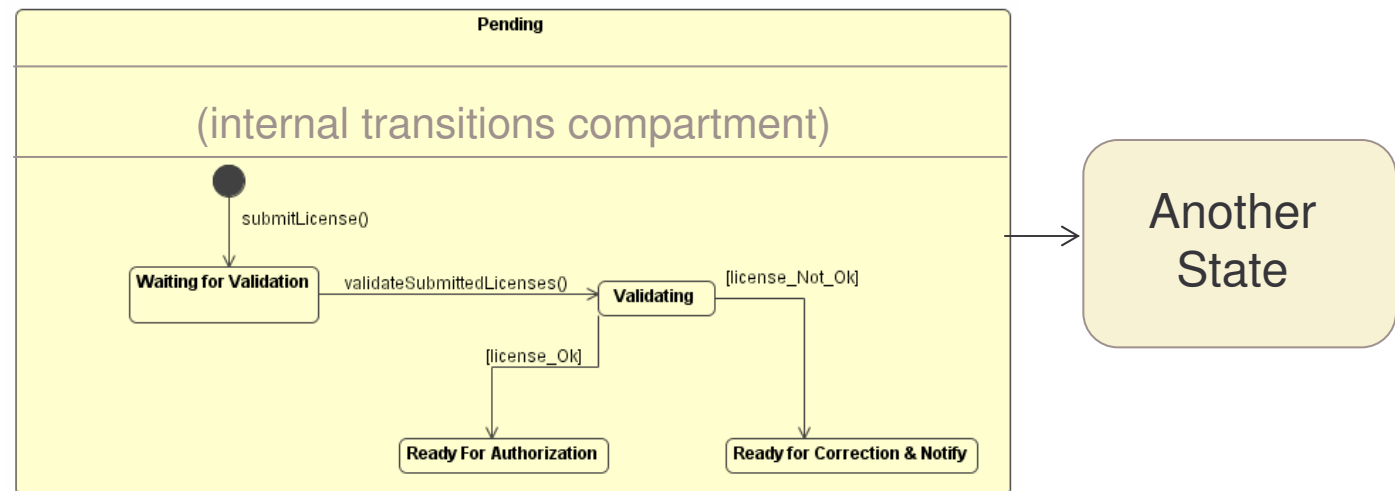


# Transitions from Sub-States 3

Alternative C:

- 1) an object may leave a super-state because all the activities in the state and its sub-state has been completed
- 2) it is called an **automatic transition**
- 3) draw the transition from the edge of the super-state to the new state

When the activities finish an **automatic transition** changes the state to **AnotherState**



# History Indicator 1

---

The history indicator is used to represent the ability for doing backtracking to a previous composite state.

Represents a pseudo-state and is a shorthand notation to solve a complex modelling problem.

May refer to:

- 1) **shallow history**: the object should return to the last sub-state on the top most layer of sub-states
- 2) **deep history**: the object needs to return to the exact sub-state from it which left, no matter how many layers down that is

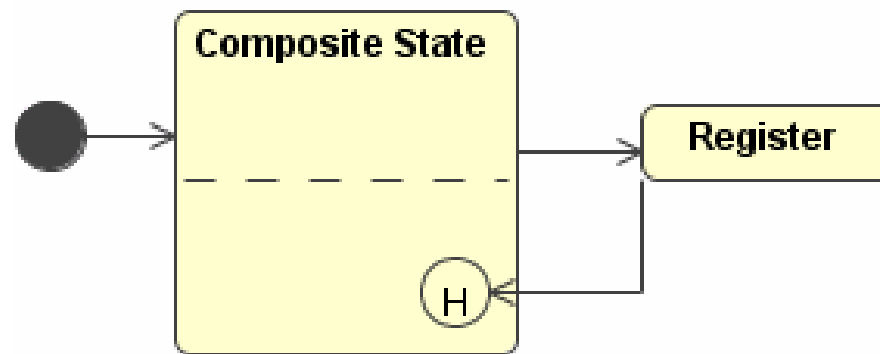
# History Indicator 2

---

Notation:

1) shallow history  $\textcircled{H}$

2) deep history  $\textcircled{H^*}$





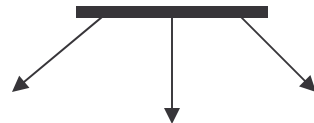
# Split of Control

---

**Split of control** means that based on a single transition it is necessary to proceed with several tasks concurrently.

Notation:

- 1) a single transition divided into multiple arrows, each pointing to a different sub-state
- 2) the division is accomplished by the synchronization bar



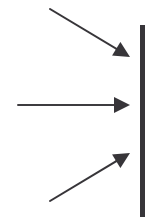
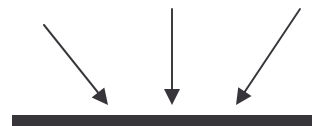
# Merge of Control

---

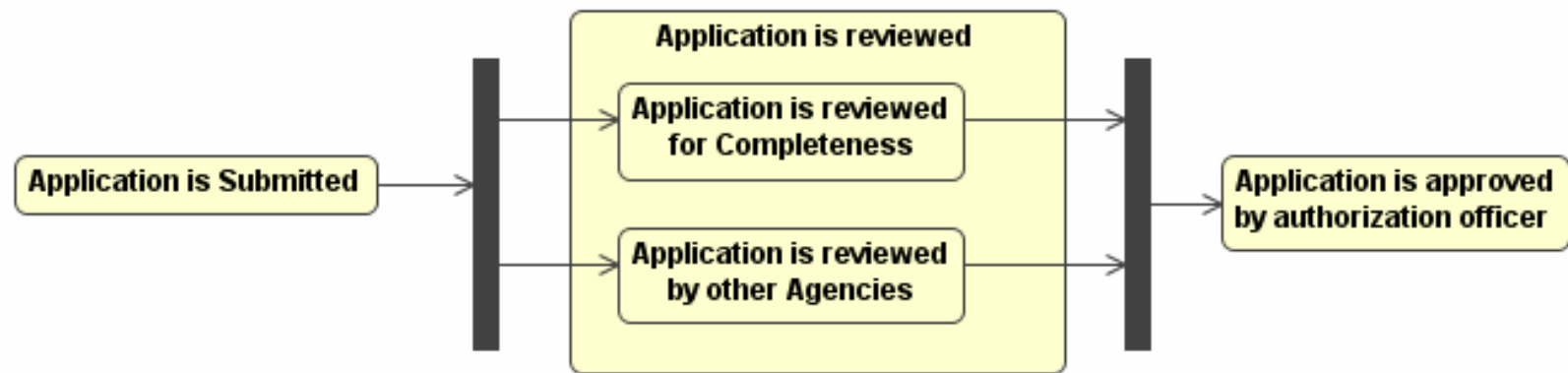
**Merge of control** means that based on the completion of a number of transitions it is necessary to proceed with a single task.

Notation:

- multiple transitions converge to a synchronization bar and only one transition outputs from the bar

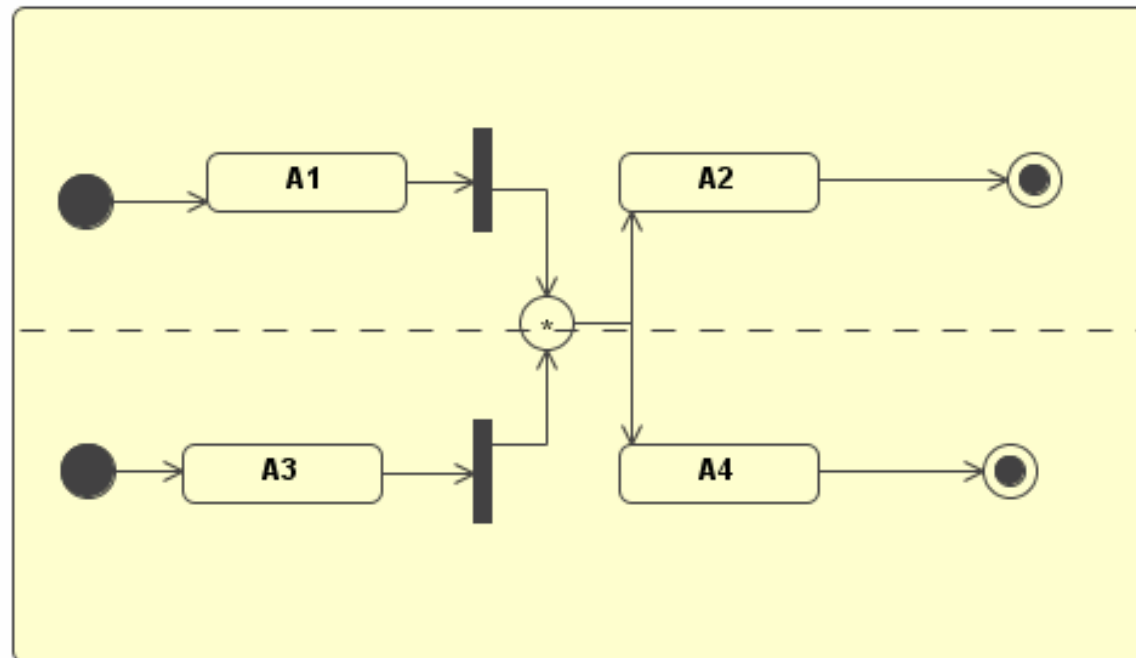


# Example: Split/Merge Control

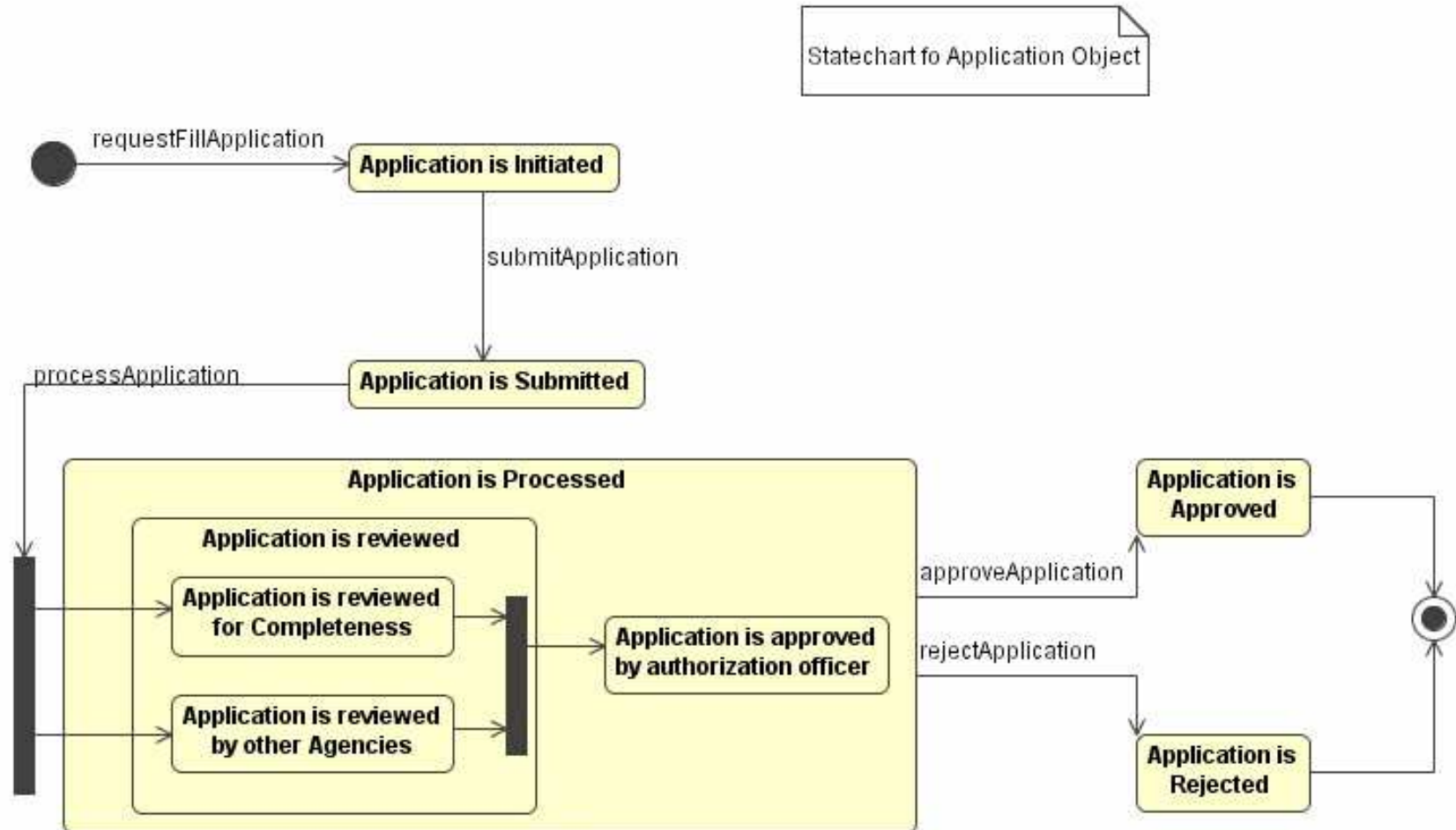


# Synchronous States

**Synchronous states** extend the split and merge of control across regions to coordinate the behaviour of concurrent states.



# Example: Case Study



# Summary

# Design Modelling

---

1) Design Concepts

2) Design Patterns

3) Design Class Diagrams

4) Design Sequence Diagrams

5) Activity Diagrams

6) Design Statechart Diagrams

7) Summary

# Summary 1

---

System design involves the transformation of analysis models of the application domain into design models of the solution space.

Object design includes the service specification for classes, component selection, object model restructuring and object model optimization.

Design patterns are partial solutions to common problems. They name, abstract and identify the key aspects of common design structure that make them useful for creating reusable object oriented design.



# Summary 2

---

There are 23 basic patterns as provided by the GoF (slide 395).

Design class diagrams provide specification for software classes and interfaces in an application.

Typical information contained in a Design Class Diagram include classes, associations, attributes, interfaces, methods, attribute type information, navigability and dependencies.

Design Sequence Diagram present the interactions between objects needed to provide a specific behaviour.

# Summary 3

---

Design sequence diagrams will specify message types (synchronous and asynchronous), temporal constraints, object creation and destruction and recursion.

Activity diagrams models the dynamic aspect of a system by showing the flow from one activity to another.

Activity diagram can be used to describe a workflow or the details of an operation.

An activity diagram may also show the flow of an object as it moves from one state to another at different points in the flow of control.

# Summary 4

---

Design statechart diagrams describes detailed the internal behavior of objects.

Design Statechart diagrams provide more implementation details with features such as sub-states, static branch points, sub-machine states etc.

# Exercise 1

---

1. Identify and list the relevant software classes for your system by analyzing the various interaction diagrams provided as part of your requirement models.
2. By using some of the information contained in the conceptual class diagrams, describe the attributes and the operation of the software classes. Indicate the types for attributes and operations of the classes as well their visibilities.
3. Present your software classes in a design class diagram and indicate navigability on class associations.
4. Add dependencies to your design class diagram.

# Exercise 2

---

5. Provide a design sequence diagram based on your software classes to describe the important interactions in your system (for example those associated with your use cases and scenarios).
6. For each of the messages specified in your interactions, indicate the type of the message (synchronous or asynchronous) and their parameters.
7. Consider the operations specified in the design classes. Provide activity diagrams to describe the details of these operations.

# Exercise 3

---

- 8) Consider five objects associated with your software or design classes, describe in details the behaviour of these objects using a design statechart.

# Implementation Model

# The Course: Overview

---

- |                             |                         |
|-----------------------------|-------------------------|
| 1) The Course               | 7) Design Model         |
| 2) Object Oriented Concepts | 8) Implementation Model |
| 3) UML Basics               | 9) Deployment Model     |
| 4) Case Study               | 10) Unified Process     |
| 5) Requirement Model        | 11) Tools               |
| 6) Architecture Model       | 12) Summary             |



# Overview

---

- 1) Implementation Phase
- 2) Implementation Model
- 3) Implementation Component Diagram
- 4) Elements in the Component Diagram:
  - a) Packages
  - b) Components and Interfaces
- 5) Modelling Techniques
- 6) Summary

# Implementation Phase

Involves the **implementation** of the design models.

It considers **non-functional requirements** and the **deployment** of the executable modules onto nodes.

Two models are developed during this phase:

- 1) the **implementation model** describing how the design elements will be implemented in terms of software system
- 2) the **deployment model** describing how the implemented software will be deployed on the physical hardware

# Implementation Model

The implementation model indicates how various aspects of the design map onto the target language.

It describes how components, interfaces, packages and files are related.

The modelling elements are:

- 1) packages
- 2) components
- 3) their relationships

and they are shown in the **implementation component diagram**.

# Packages

---

A package represents a **physical partitioning** of the system.

Packages are organized in a **hierarchy of layers**.

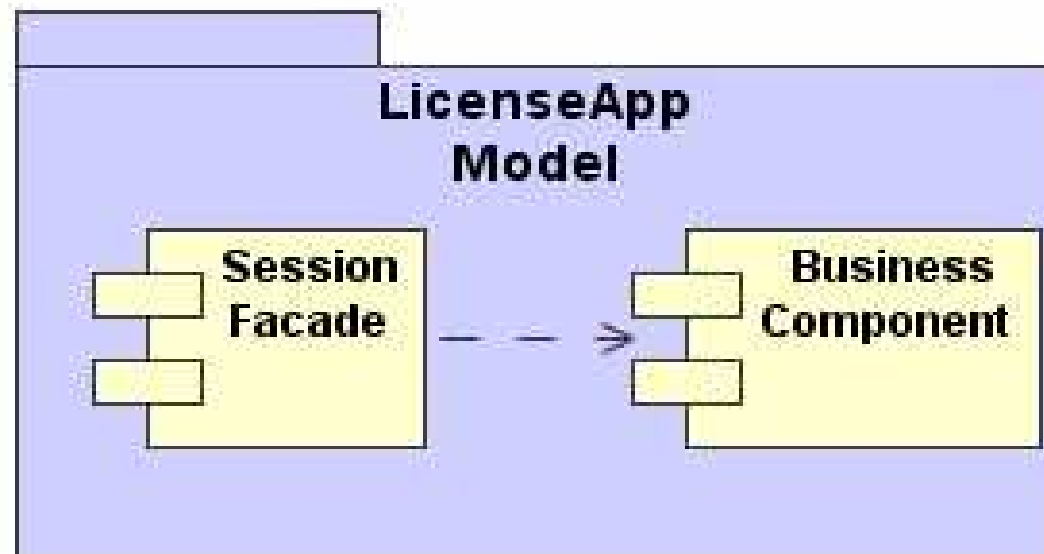
Packages are useful for:

- 1) associating related components and interfaces
- 2) resolving naming problems
- 3) providing some privacy for classes, attributes, and operations that should not be visible outside the package

# Example: Package

The package LicenseApp Model has two components:

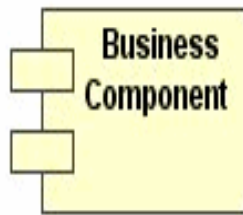
- 1) SessionFacade
- 2) BusinessComponent



# Components

---

A component:



- 1) is a **physical part** of a system
- 2) is **replaceable**
- 3) conforms to and **provides the realization** of a set of interfaces

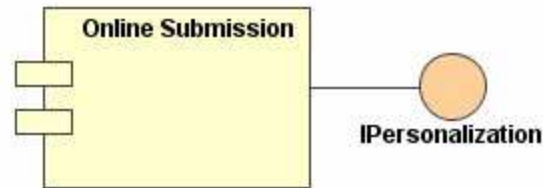
Components are **grouped in packages**.

Components **can be organized by** specifying dependency, generalization, association, aggregation, and realization **relationships** among them.

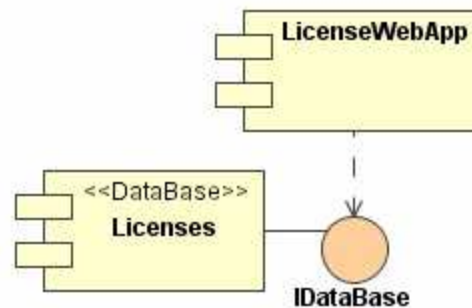
Component modelling is very important to control the versioning and configuration management as system evolves.

# Components and Interfaces

The component that realizes an interface is connected to it using a full realization relationship.



The component that accesses the services of other component through the interface is connected to the interface using a dependency relationship.



# Modelling Techniques

Component diagrams are used to model the static implementation view of a system.

To model this view, it is possible to use component diagrams in one of four ways:

- 1) to model source code
- 2) to model executable releases
- 3) to model physical databases
- 4) to model adaptable systems



# Modelling Source Code

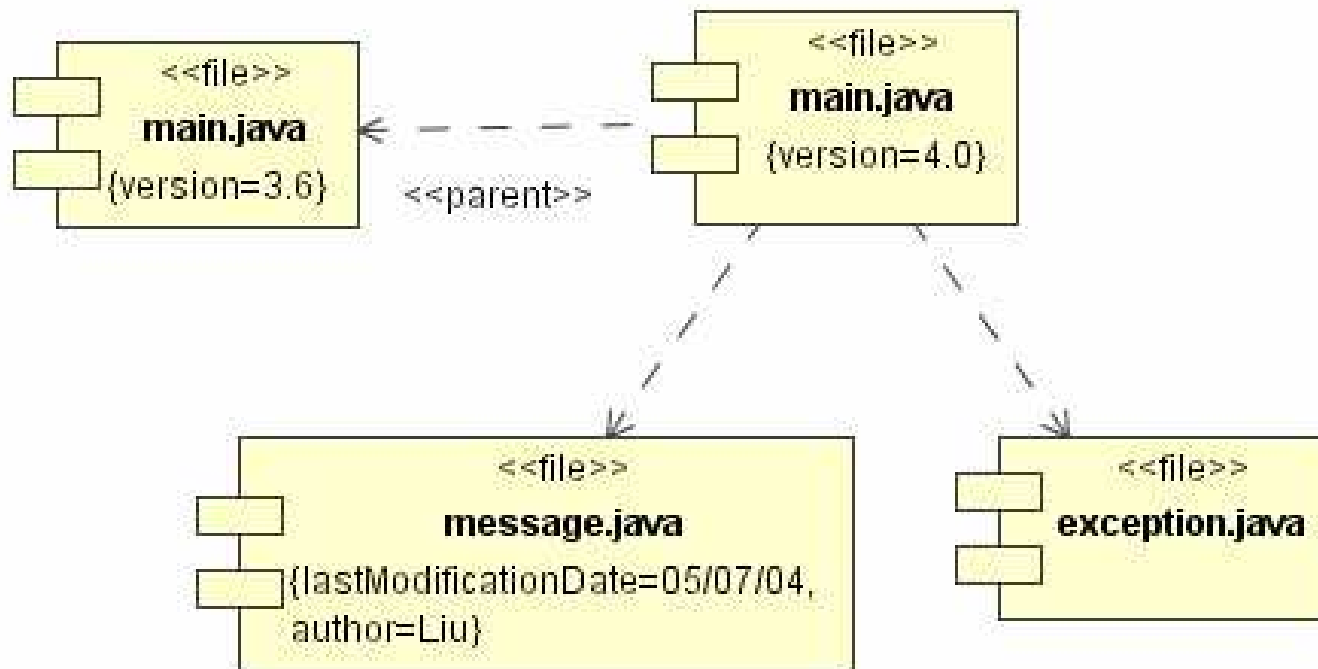
---

Component diagrams are used to model the configuration management of source files, which represent work-product components.

Modelling procedure:

- 1) identify the set of source code files of interest, either by forward or reverse engineering, and model them as components stereotyped as **files**
- 2) for larger systems, use packages to show groups of source code files
- 3) consider exposing a tagged value to show interesting information such as author, version number, etc.
- 4) model the compilation dependencies among these files using dependencies

# Example: Source Code Files



# Modelling Executable Releases

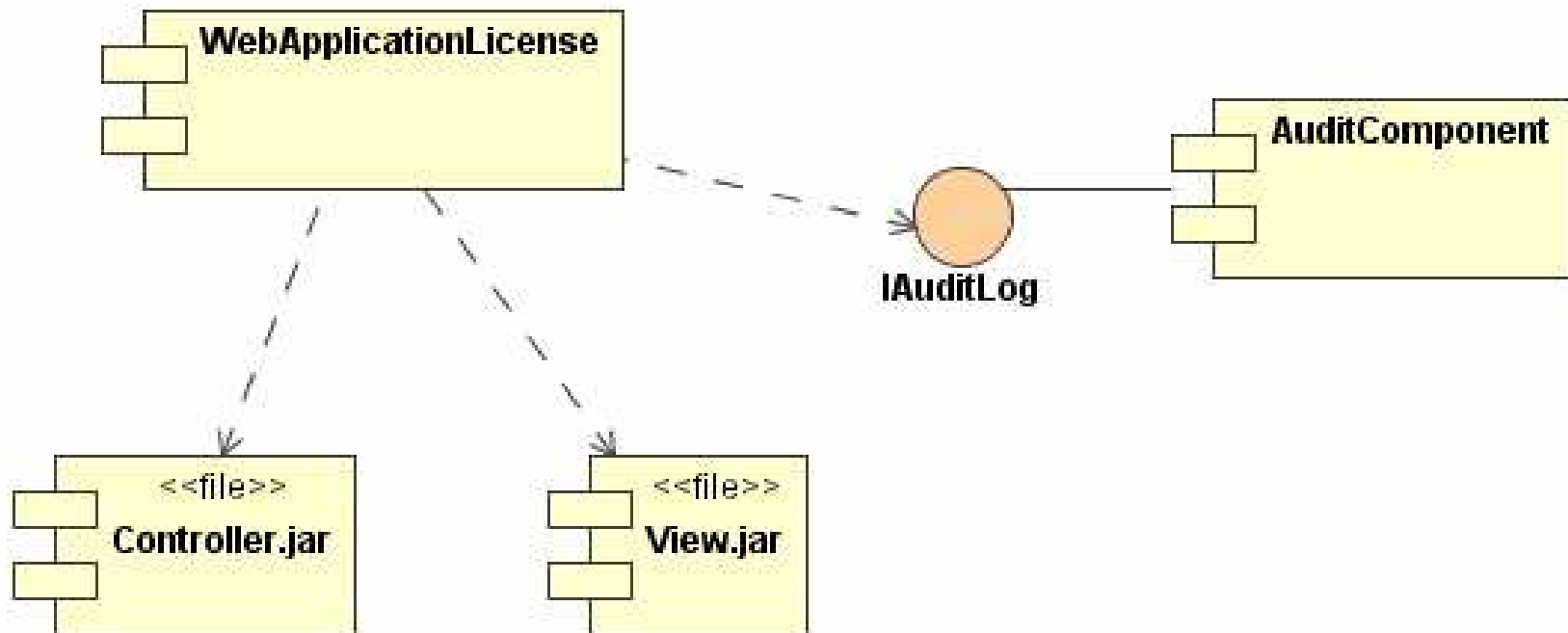
Component diagrams are used to visualize, specify, construct, and document the configuration of the executable releases, including the deployment components that form each release, and the relationships among those components.

Each component diagram should focus on one set of components at a time, such as all components that live on one node.

Modelling procedure:

- 1) identify the set of components to model
- 2) consider the stereotype of each component in the set
- 3) for each component, consider its relationship to its neighbors

# Example: Executables



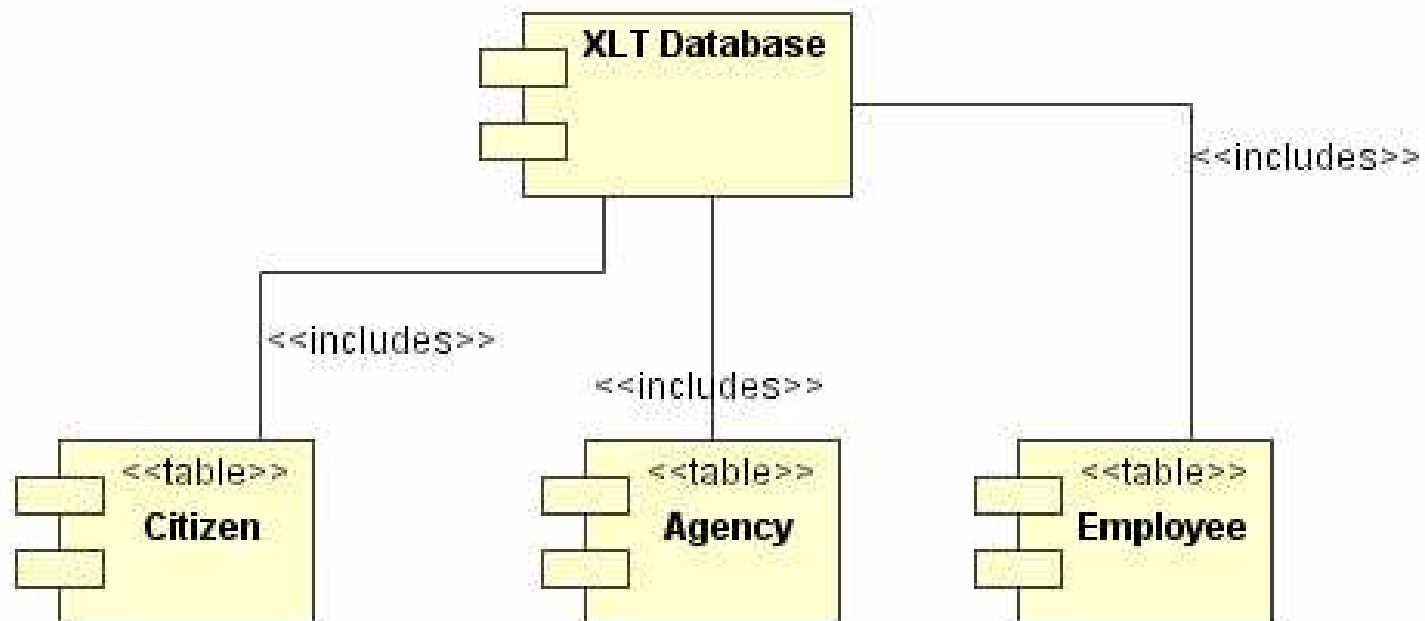
# Modelling Physical Databases

Component diagrams can be used to visualize, specify, construct and document the mapping of classes into tables of a database.

Modelling procedure:

- 1) identify the classes in your model that represent your logical database schema
- 2) select a strategy for mapping these classes to tables, possibly considering the physical distribution
- 3) create a component diagram containing components stereotyped as **tables** to model the mapping

# Example: Physical Databases



# Modelling Adaptable Systems 1

All the component diagrams shown were used to model static views.

A **static view** means that its components spend their entire life on one node.

In some distributed systems is necessary to model dynamic views.

A **dynamic view** models components migrating from one node to another.

To model a dynamic view we need a combination of:

- 1) component diagrams
- 2) object diagrams
- 3) interaction diagrams

# Modelling Adaptable Systems 2

Modelling procedure:

- 1) consider the physical distribution of the components that may migrate from node to node
- 2) specify the location of a component instance by marking it with a tagged value, which you can then render in a component diagram
- 3) model the actions that cause a component to migrate creating a corresponding interaction diagram that contains component instances



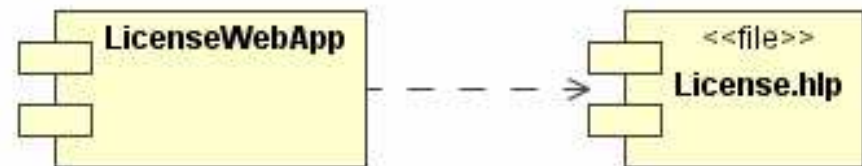


# Tables, Files and Documents

Implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files.

Modelling procedure:

- 1) identify the auxiliary components that are part of the physical implementation
- 2) model these things as components (introduce new stereotypes as needed)
- 3) model the relationships among these auxiliary components and other executables.



# APIs

---

An Application Programming Interface (API) is essentially an interface that is realized by one or more components.

One concept, two perspectives:

- 1) as developer: interested only in the interface
- 2) as system configuration management: interested in which component realizes the interface

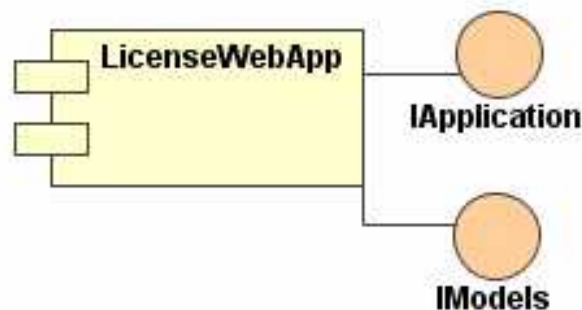
The operations associated with any semantically rich API will be fairly extensive, and mostly it is not needed to visualize them. Instead, interfaces grouping these operations are modelled.

# Modelling an API

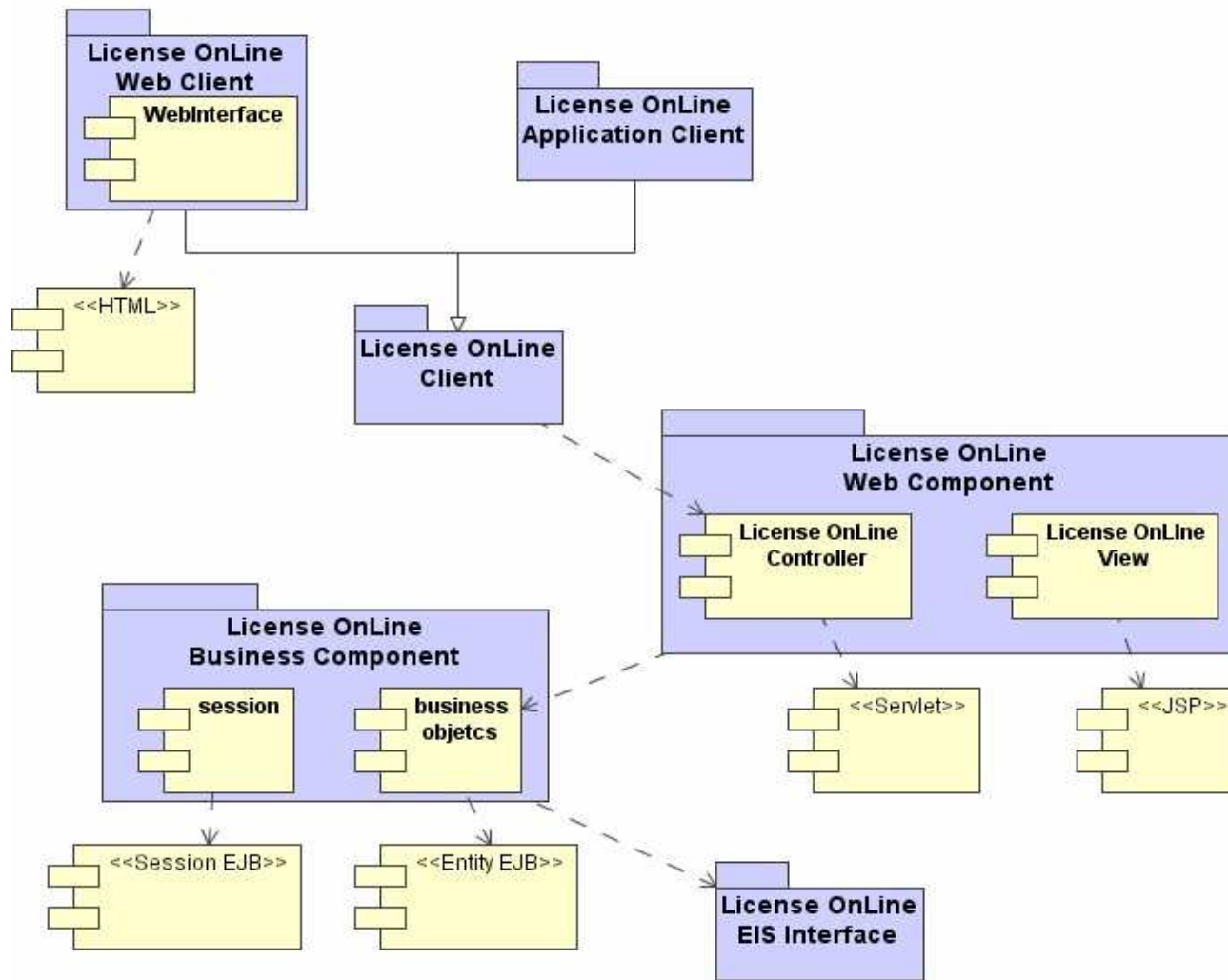
---

Procedure:

- 1) identify the programmatic seams and model each as an interface, collecting the corresponding attributes and operations
- 2) expose only those properties of the interface that are important to visualize in the given context
- 3) model the realization of each API only if it is important to show the configuration of a specific implementation



# Example: Case Study



# Summary

---

The implementation component diagram describes how components, interfaces, packages and files are related.

Implementation component diagrams may be used to model source code, executable releases, physical databases and adaptable systems.

# Deployment Model

# The Course: Overview

---

- |                             |                         |
|-----------------------------|-------------------------|
| 1) The Course               | 7) Design Model         |
| 2) Object Oriented Concepts | 8) Implementation Model |
| 3) UML Basics               | 9) Deployment Model     |
| 4) Case Study               | 10) Unified Process     |
| 5) Requirement Model        | 11) Tools               |
| 6) Architecture Model       | 12) Summary             |

# Overview

---

- 1) deployment diagrams
- 2) nodes
- 3) nodes and components
- 4) common uses of deployment diagrams



# Deployment Diagrams

A deployment diagram shows the configuration of run-time processing **nodes** and the **components** that live on them.

They are used to model the distribution, delivery, and installation of the parts that make up the physical system.

It involves modelling the topology of the hardware on which the system executes.

Deployment diagrams are essentially class diagrams that focus on a system's nodes, and include:

- 1) nodes
- 2) dependencies and associations relationships
- 3) components
- 4) packages

# Nodes

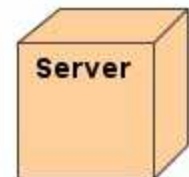
---

Nodes are used to model the topology of the hardware on which the system executes.

The components developed or reused must be deployed on some set of hardware in order to execute.

Nodes represent the hardware on which these components are deployed and executed.

UML notation for nodes allows for visualizing a node independently of any specific hardware.



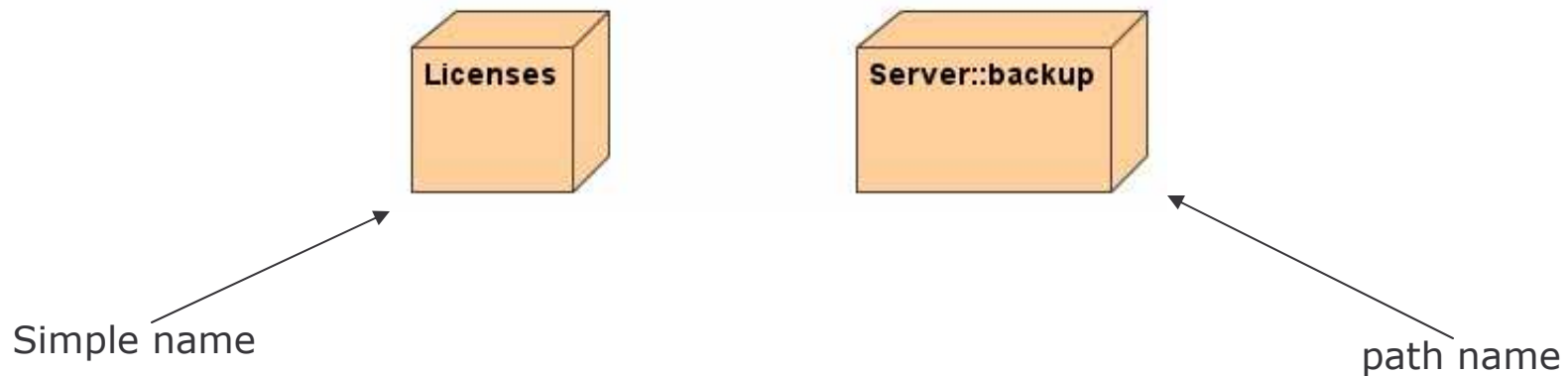
Stereotypes allows to represent specific kinds of processors and devices.

# Nodes Names

---

Every node must have a name that distinguishes it from other nodes.

A name is a textual string which may be written as a simple name or as a path name.



# Nodes and Components

---

## Components

- 1) participate in the execution of a system.
- 2) represent the physical packaging of otherwise logical elements

## Nodes

- 1) execute components
- 2) represent the physical deployment of components

The relationship **deploys** between a node and a component can be shown using a **dependency relationship**.

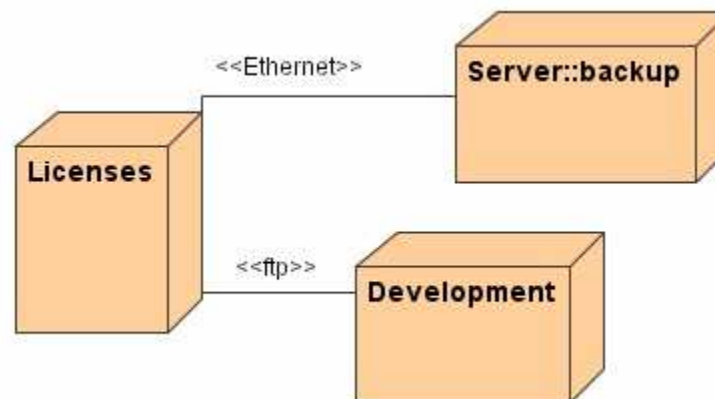
A set of objects or components that are allocated to a node as a group is called a **distribution unit**.

# Organizing Nodes

Nodes can be organized:

- 1) in the same manner as classes and components
- 2) by specifying dependency, generalization, association, aggregation, and realization **relationships** among them.

The most common kind of relationship used among nodes is an **association** representing a physical connection among them.

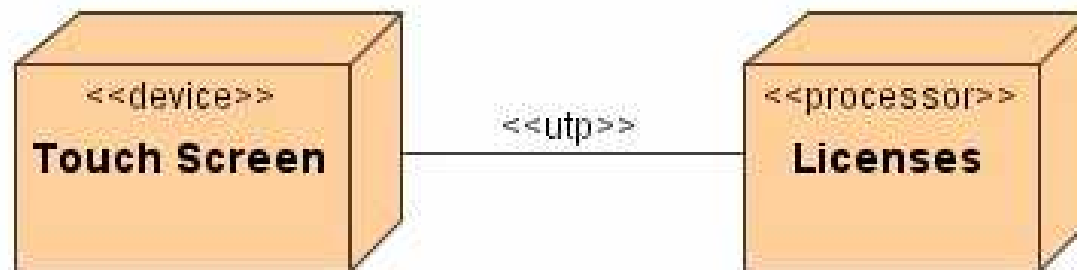


# Processors and Devices

---

A **processor** is a node that has processing capability. It can execute a component.

A **device** is a node that has no processing capability (at least at the level of abstraction showed).



# Modelling Nodes

---

Procedure:

- 1) identify the computational elements of the system's deployment view and model each as a node
- 2) add the corresponding stereotype to the nodes
- 3) consider attributes and operations that might apply to each node.

# Distribution of Components

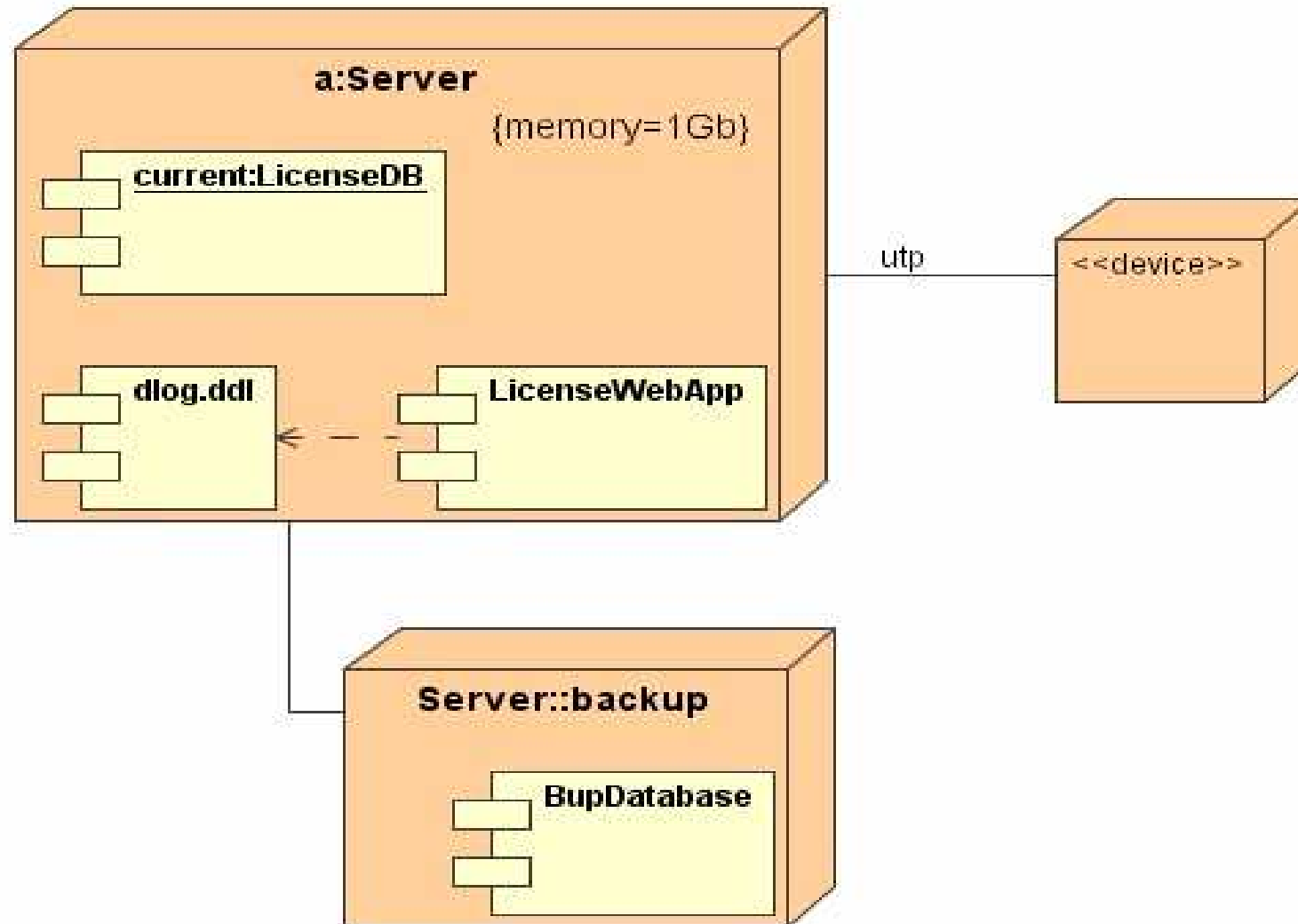
To model the topology of a system it is necessary to specify the physical distribution of its components across the processors and devices of the system.

Procedure:

- 1) allocate each component in a given node
- 2) consider duplicate locations for components, if it is necessary
- 3) render the allocation in one of these ways:
  - a) don't make visible the allocation
  - b) use dependency relationship between the node and the component it's deploy
  - c) list the components deployed on a node in an additional compartment



# Example: Deployment Diagram



# Common Uses

---

Deployment diagrams may be used to model:

- 1) **embedded systems**: software-intensive collection of hardware that interfaces with the physical world. Involve software that controls devices, and that in turn is controlled by external stimuli
- 2) **client/server systems**: architectures making a clear distinction between system's user interface (client) and system's persistent data (server)
- 3) **distributed systems**: encompass multiple levels of servers.

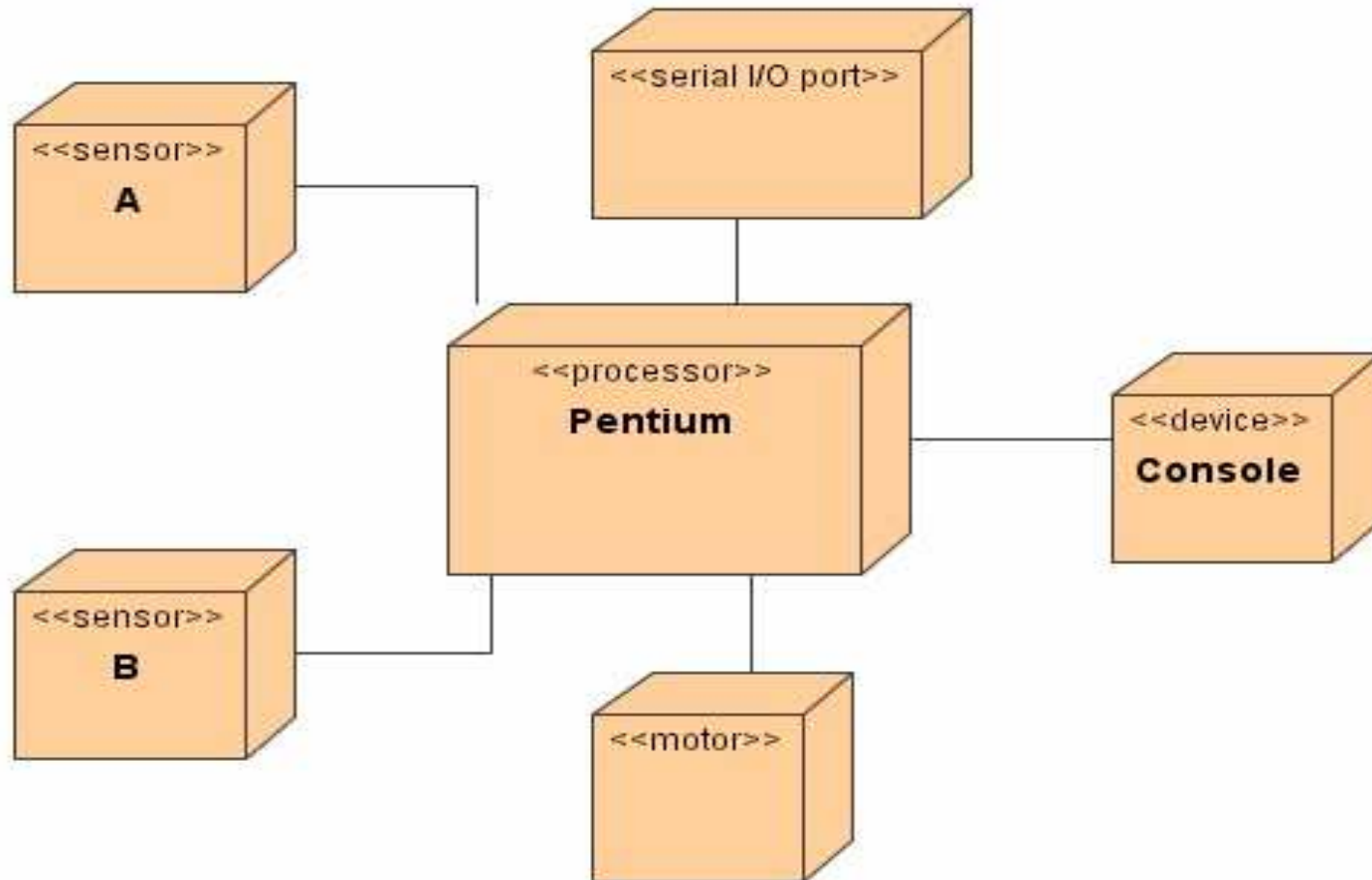
# Embedded System

---

Modelling procedure:

- 1) identify the devices and nodes that are unique to the system
- 2) provide visual indication for unusual devices with system-specific stereotypes and appropriate icons, distinguishing processors and devices
- 3) model the relationships among these processors and devices using a deployment diagram and specify the relationship between the components in the implementation view and the nodes in the deployment view
- 4) if necessary, expand on any intelligent devices by modelling their structure with a more detailed deployment diagram.

# Example: Embedded System

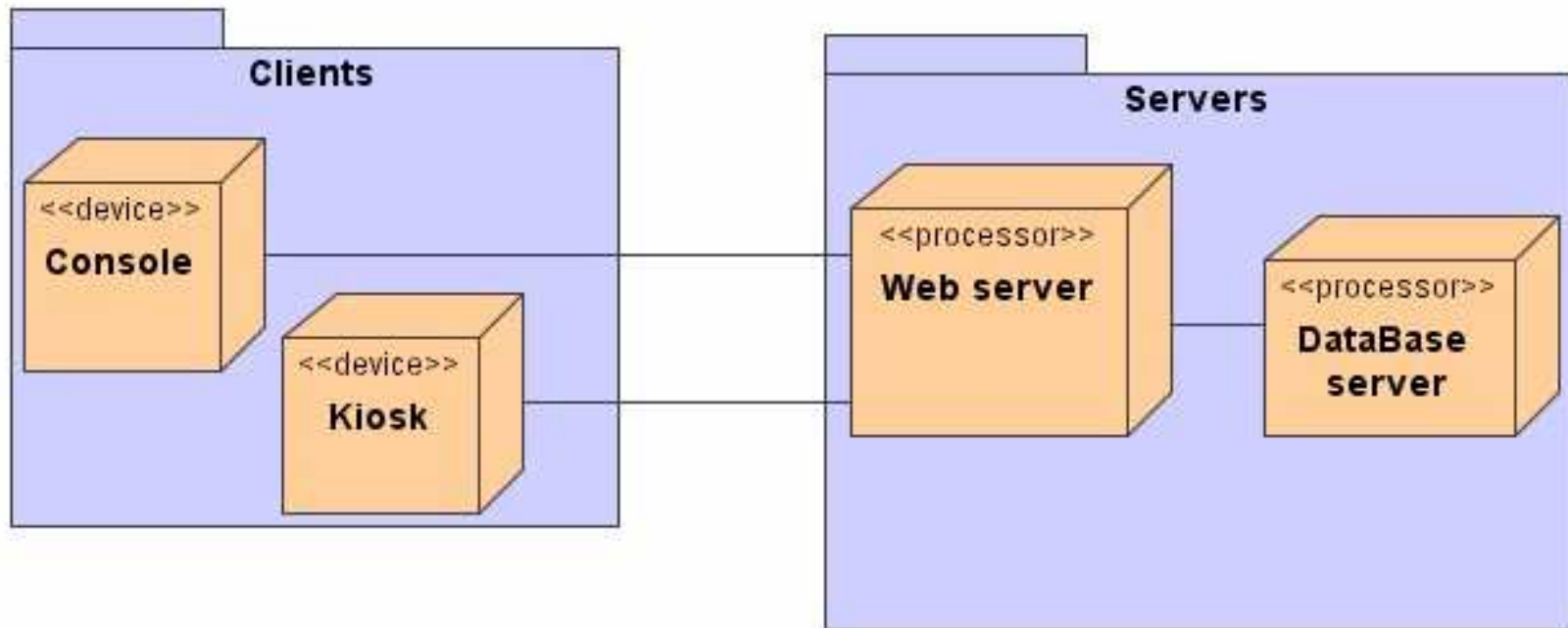


# Client/Server Systems

Modelling procedure:

- 1) identify the nodes that represent the system's client and servers processors
- 2) highlight those devices that are relevant to the system
- 3) provide visual indication for these devices with stereotypes
- 4) model the topology of these nodes in a deployment diagram, and specify the relationship between the components in the implementation view and the nodes in the deployment view.

# Example: Client/Server System



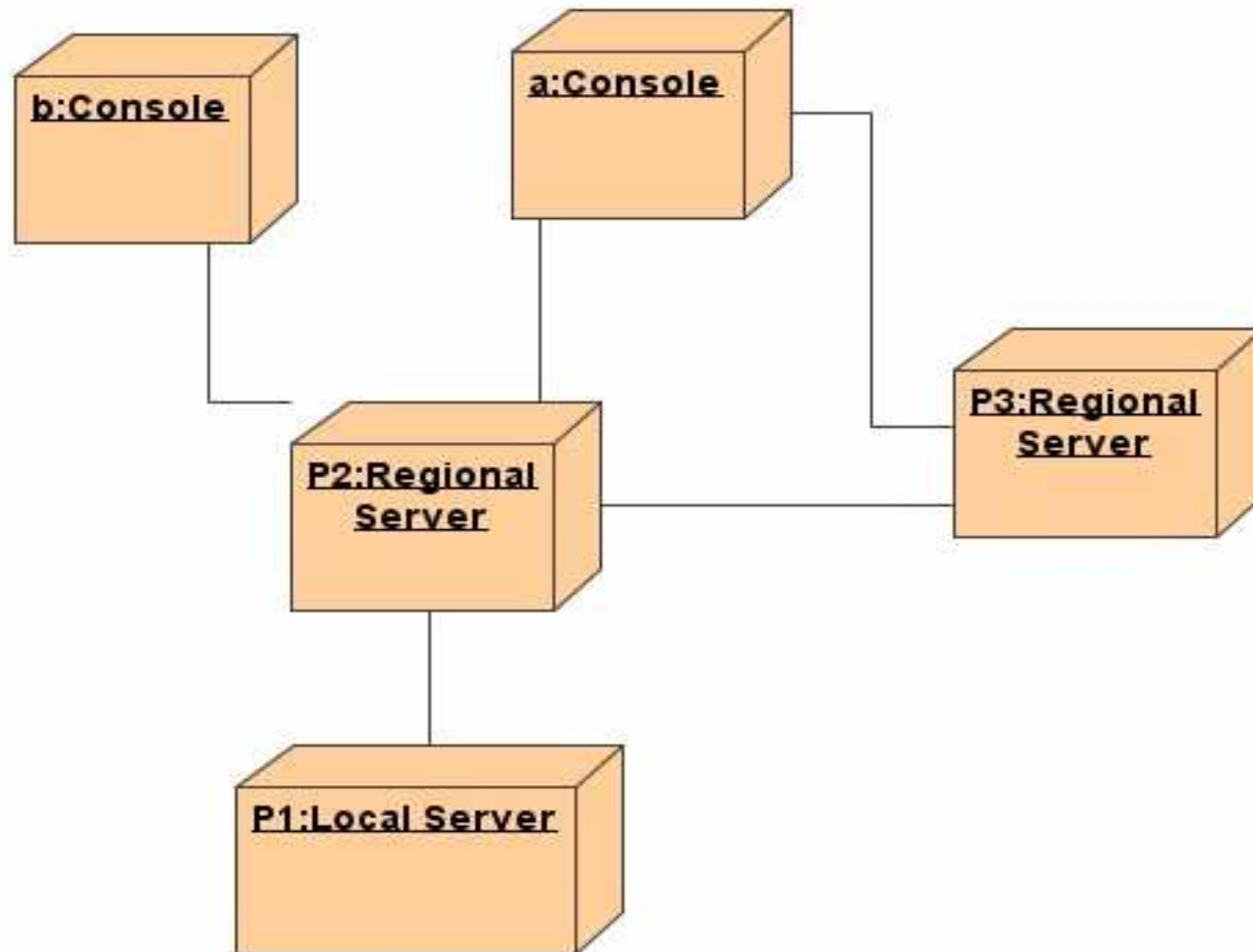
# Distributed Systems

---

Modelling procedure:

- 1) identify the system's devices and processors
- 2) model the communication devices including sufficient detail if you need to assess the performance of the system's network or the impact of changes to the network
- 3) pay close attention to logical groupings of nodes that can be specified by using packages
- 4) model these devices and processors using deployment diagrams
- 5) if it is needed to focus on the dynamics of the system, introduce use case diagrams to specify the kinds of behaviour of interest, and expand these diagrams with interaction diagrams.

# Example: Distributed System





# Summary

---

Deployment diagrams model the topology of the hardware on which the system executes and the software installed on each of the hardware components.

Deployment diagrams include nodes, packages, components and their relationships.

Deployment diagrams may be used to model different kind of systems such as embedded, client-server or distributed systems.

# Exercise 4

---

1. Consider your architectural design. Describe the components that are likely implement the layers of your architecture.
2. Specify the nature of the work product components or artifacts that are likely to implement these components.
3. Describe the relationship between the components in question 1 and the work products identified in question 2 using an implementation diagram.
4. Describe a simple database schema for your system.

# Exercise 5

---

- 5) Describe an ideal deployment environment for your system.
- 6) Specify how the various components of your systems identified in question 3 will be distributed in the environment described in question 5.

# Unified Process

# The Course: Overview

---

- |                             |                         |
|-----------------------------|-------------------------|
| 1) The Course               | 7) Design Model         |
| 2) Object Oriented Concepts | 8) Implementation Model |
| 3) UML Basics               | 9) Deployment Model     |
| 4) Case Study               | 10) Unified Process     |
| 5) Requirement Model        | 11) Tools               |
| 6) Architecture Model       | 12) Summary             |

# Overview

---

- 1) Software Development Process
- 2) Unified Process Overview
- 3) Unified Process Structure
  - 1) Building Blocks
  - 2) Phases
  - 3) Workflows

# Why a process ?

A process defines:

- 1) **who** is doing **what**
- 2) **when** to do it
- 3) **how** to reach a certain goal
- 4) the inputs and outputs for each activity



# Development Process

Is a **framework** which guides the tasks, people and define output products of the development process.

It is a framework because:

- 1) provides the inputs and outputs of each activity
- 2) does not restrict how each activity must be performed
- 3) must be tailored for every project

There is **no universal process**.



# Unified Process - Overview

Key elements:

- 1) iterative and incremental
- 2) use case-driven
- 3) architecture-centric

# Iterative and Incremental 1

The design process is based on iterations that address different aspects of the design process.

The iterations evolve into the final system (incremental aspect).

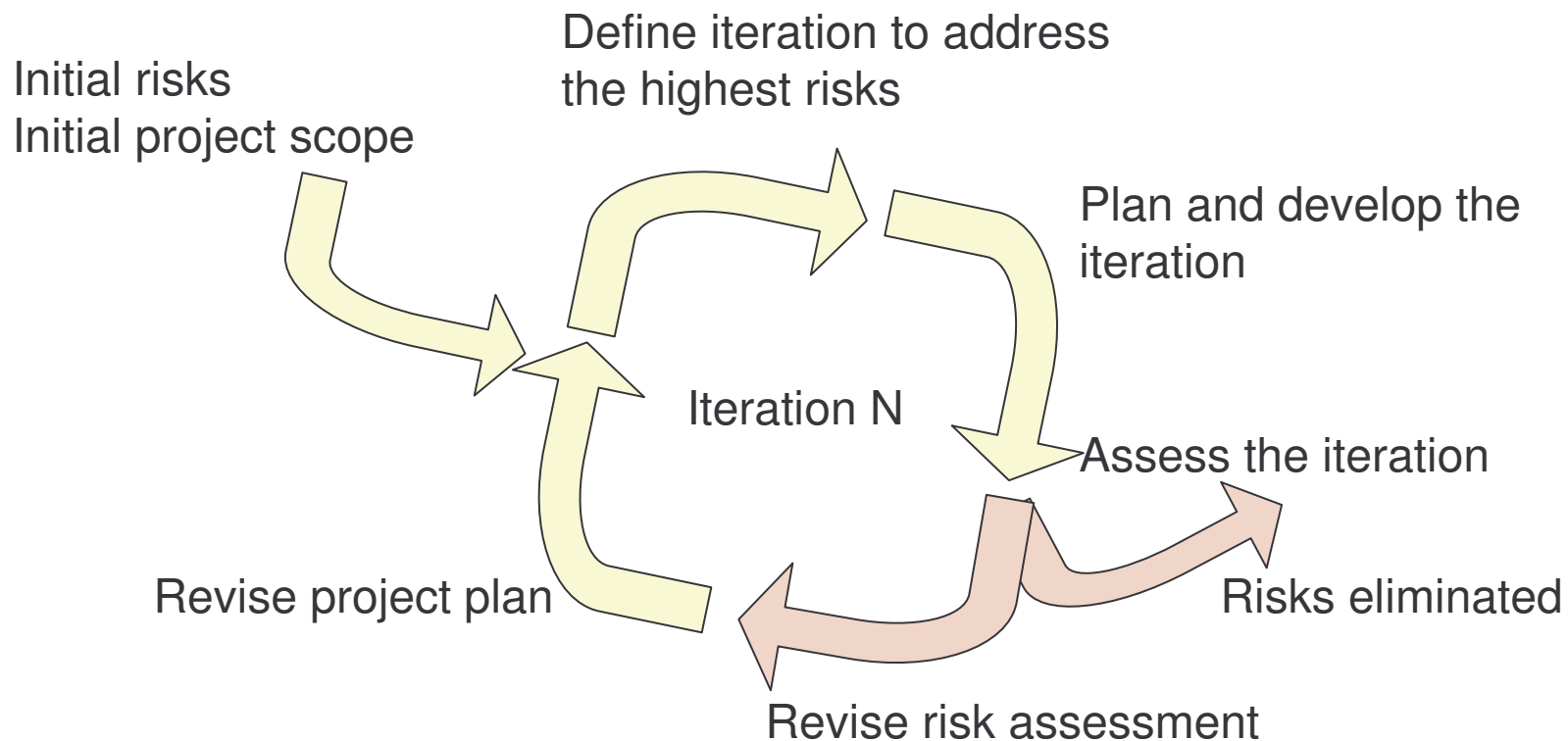
The process does not try to complete the whole design task in one go.

How to do it?

- 1) plan a little
- 2) specify, design and implement a little
- 3) integrate, test and run
- 4) obtain feedback before next iteration

# Iterative and Incremental 2

Technical risks are assessed and prioritized early and are revised during each iteration.



# Use Case-Driven

---

Use cases are used for:

- 1) identify users and their requirements
- 2) aid in the creation and validation of the architecture
- 3) help produce definitions of test cases and procedures
- 4) direct the planning of iterations
- 5) drive the creation of user documentation
- 6) direct the deployment of the system
- 7) synchronize the content of different models
- 8) drive traceability throughout models

# Architecture-Centric

---

## Problem:

- with the iterative and incremental approach different development activities are done concurrently

## Solution:

- the system's architecture ensures that all parts fit together

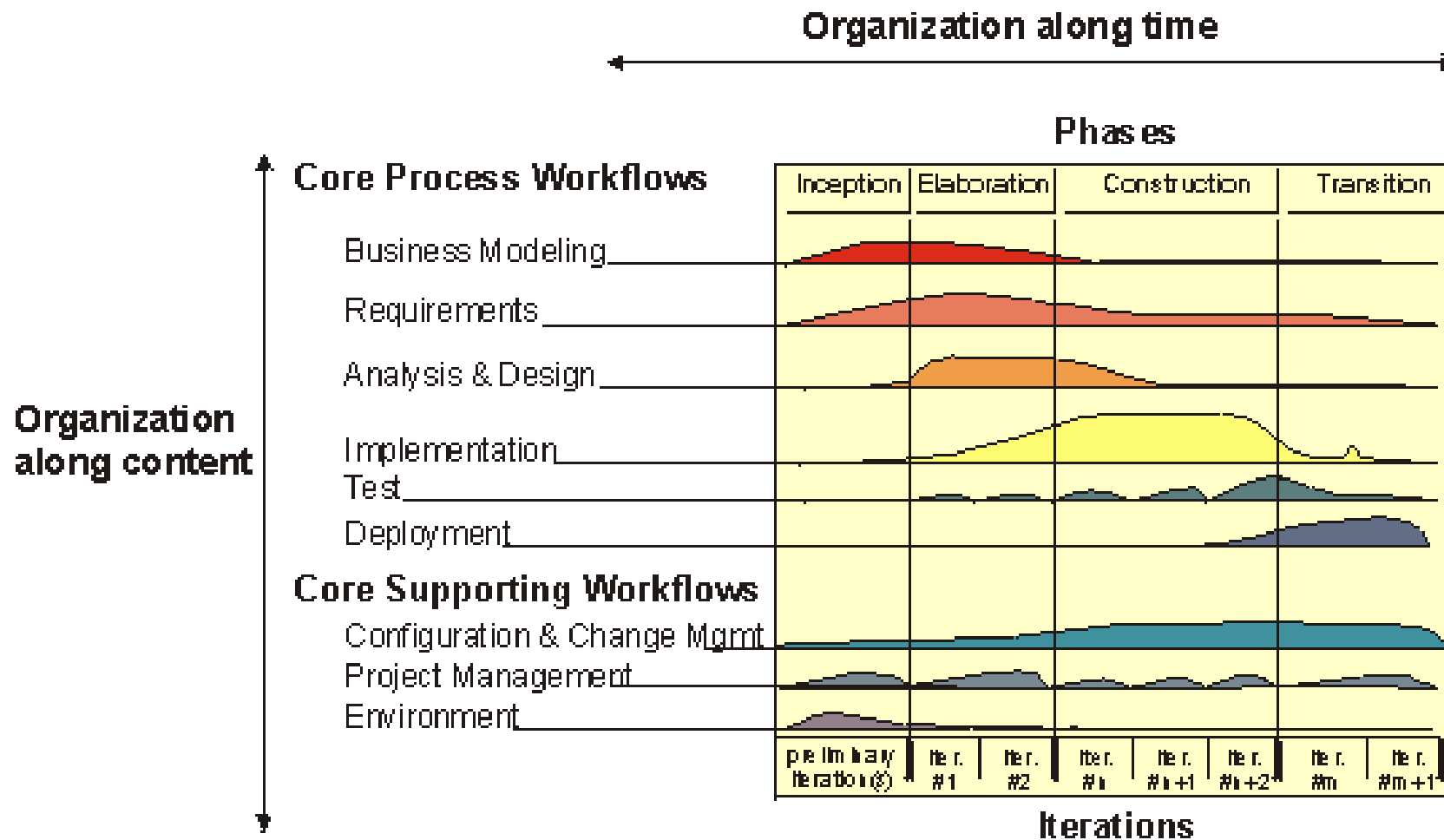
"An architecture is the skeleton on which the muscles (functionality) and skin (user-interface) of the system will be hung".

# Unified Process - Structure

Two dimensions:

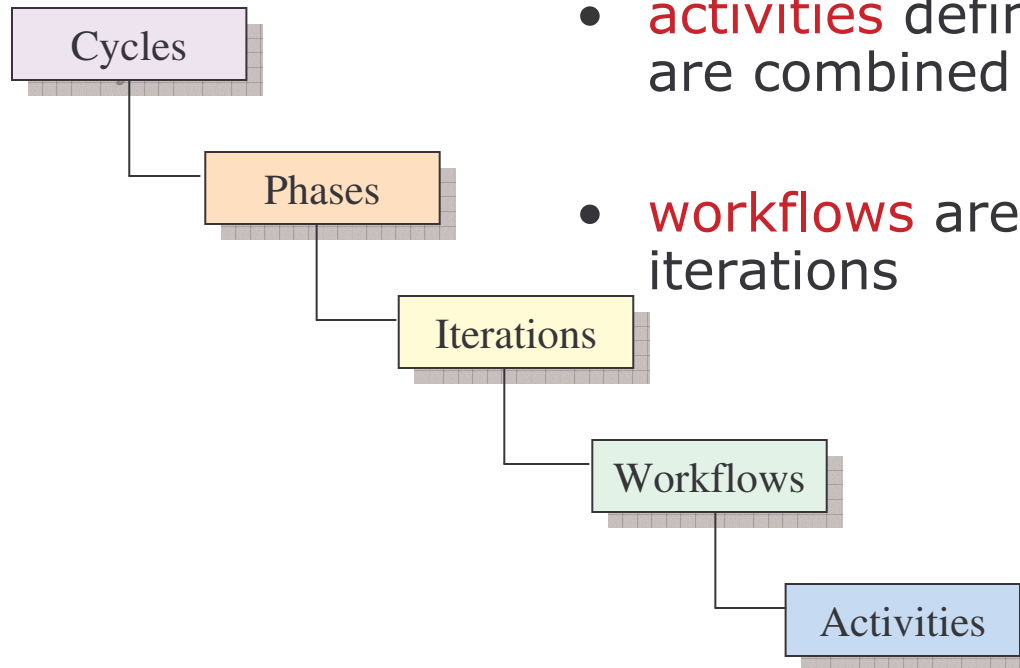
- 1) **time** —————→ division on the life cycle into **phases** and iterations
  
- 2) **process components** —→ production of a specific set of artifacts with well-defined activities called **workflows**

# The Development Process



# Building Blocks

---



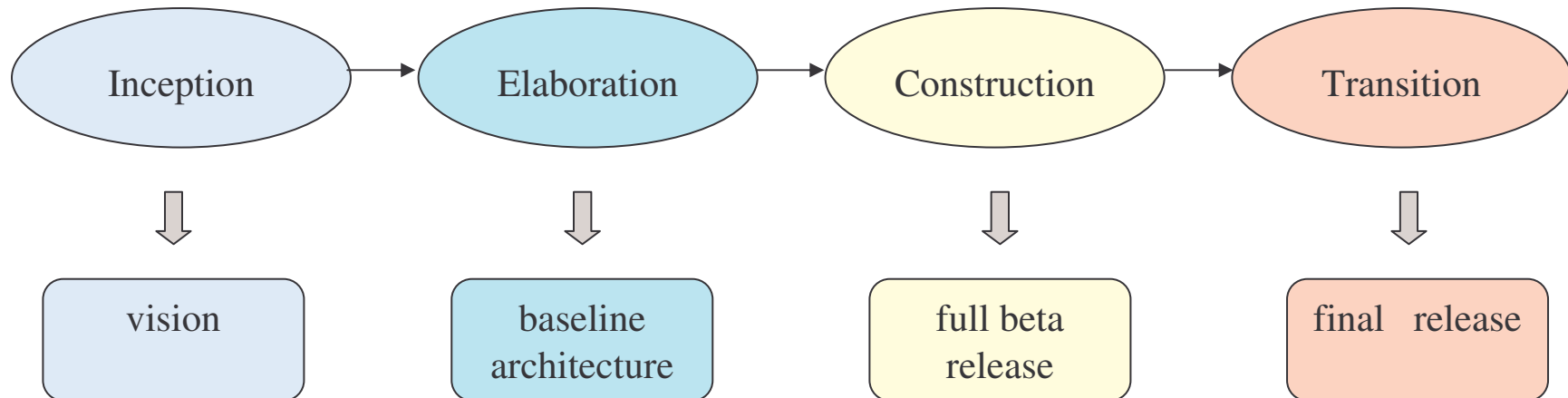
- **activities** define detailed work and are combined in workflows
- **workflows** are organized into iterations

- each **iteration** identifies some aspect of the system and are organized into phases
- **phases** can be grouped into cycles
- **cycles** focus on the generation of successive releases



# Life Cycle Phases

Phases and major deliverables of the Unified Process



# Inception

---

Focuses on the generation of the business case that involves:

- 1) identification of core uses cases
- 2) definition of the actual scope
- 3) identification of risky difficult parts of the system

The main objectives are:

- 1) to prove the feasibility of the system to be built
- 2) to determine the complexity involved in order to provide reasonable estimates

Outputs:

- 1) the vision of the system
- 2) very simplified use case model
- 3) tentative architecture
- 4) risks identified
- 5) plan for the elaboration phase

# Elaboration

---

Involves:

- 1) understanding how requirements are translated into the internals of the system
- 2) producing the baseline architecture
- 3) capturing the majority of the use cases
- 4) exploring further the risks identified earlier and identifying the most significant
- 5) specifying any non-functional requirements specially those related to reliability and performance

Outputs:

- 1) **the system's architecture**
- 2) detailed use case model
- 3) set of plans for the construction phase

# Construction

---

Involves:

- 1) completing the analysis of the system
- 2) performing the majority of the design and the implementation

Outputs:

- 1) **implemented software product as a full beta release.**  
It may contain some defects
- 2) associated models

An important aspect for the success of this phase is to monitor the critical aspects of the projects, specially significant risks.

# Transition

---

Involves:

- 1) deployment of the beta system
- 2) monitoring user feedback and handling any modifications or updates required

Output:

- 1) the formal release of the software

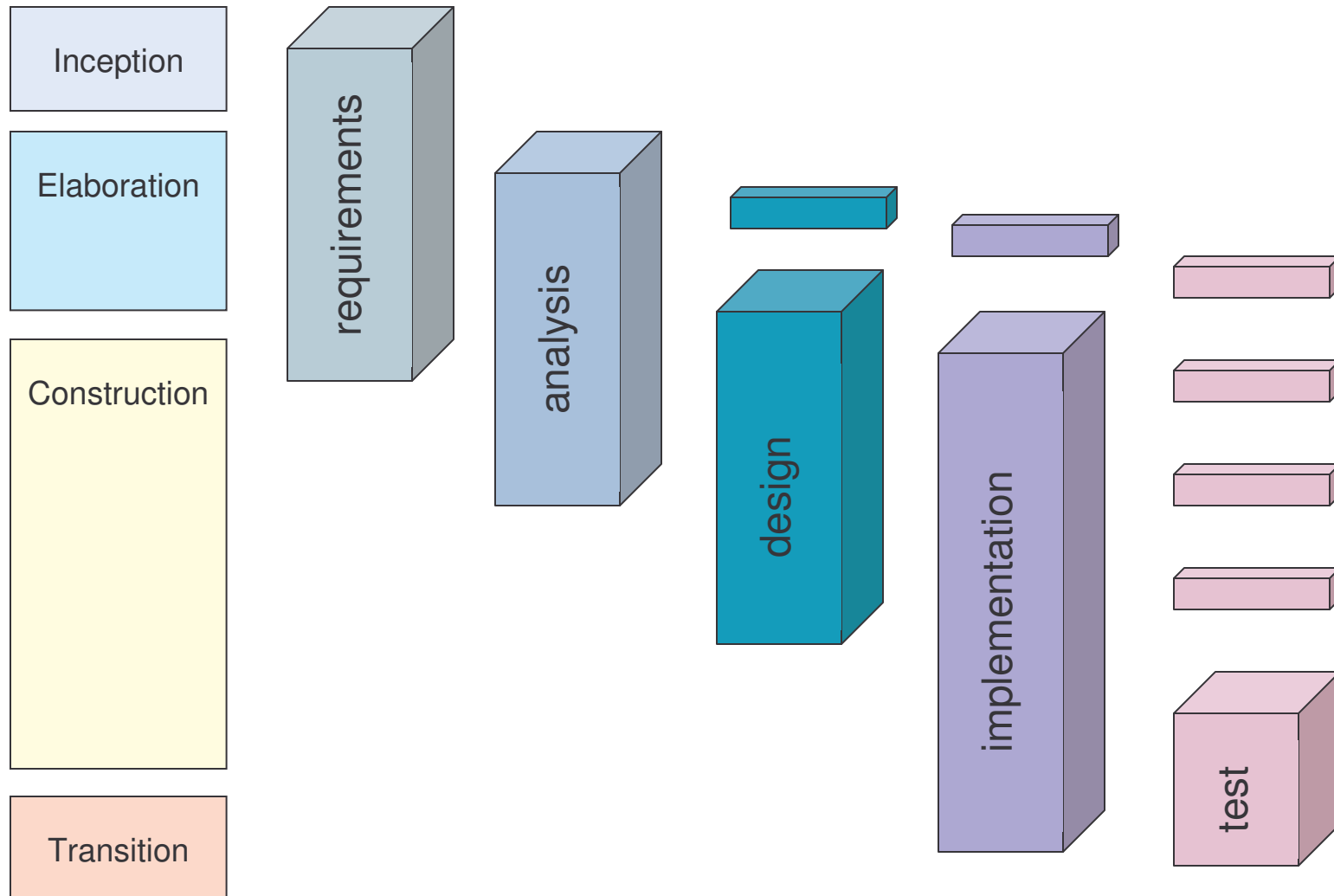
# UP - Workflows 1

---

- 1) **requirements**: focuses on the activities which allow to identify functional and non-functional requirements
- 2) **analysis**: restructures the requirements identified in terms of the software to be built
- 3) **design**: produces a detailed design
- 4) **implementation**: represents the coding of the design in a programming language, and the compilation, packaging, deployment and documentation of the software
- 5) **test**: describes the activities to be carried out for testing

# Workflows and Phases

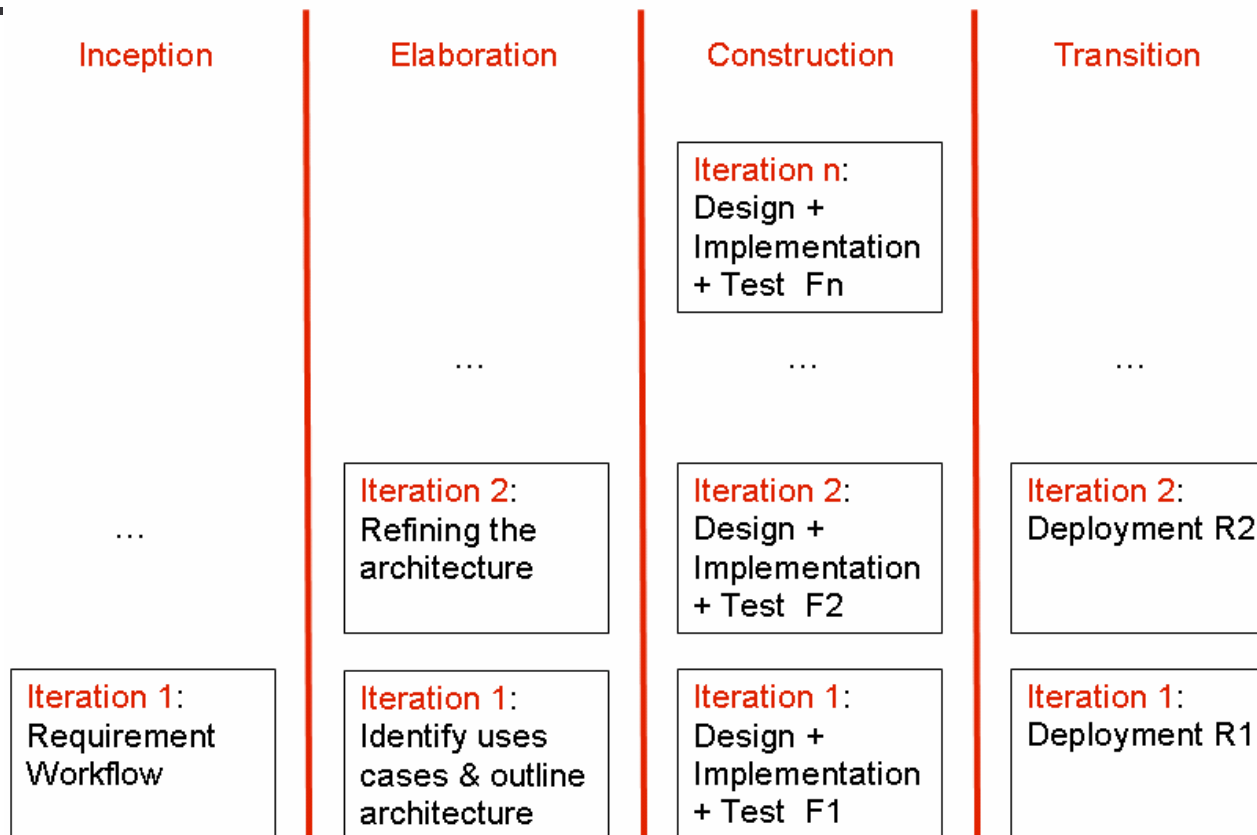
---



# Phases and Iterations

---

The iterations of workflows occur one or more times during a phase.





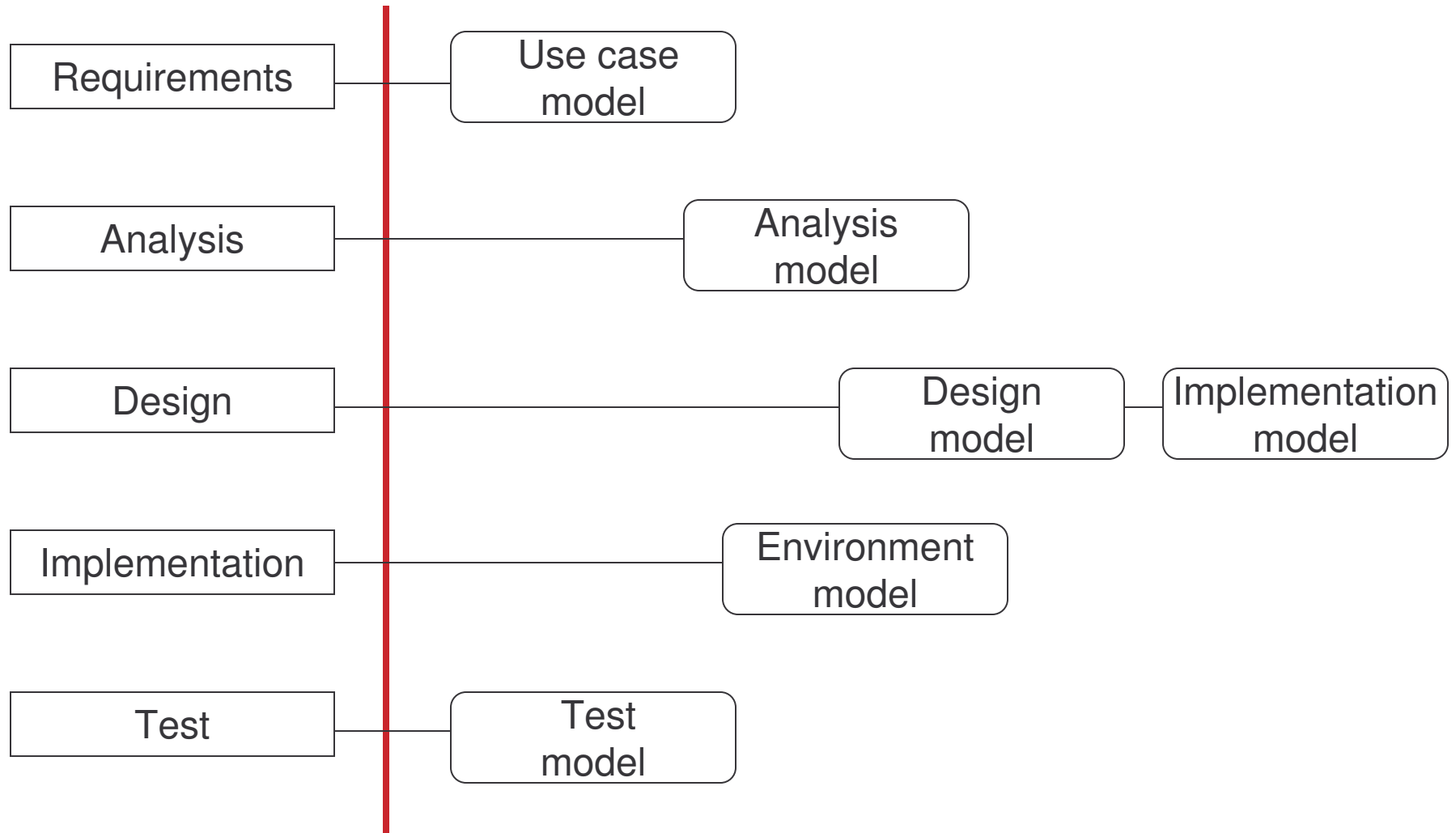
# Workflows and Activities

---

Workflows	Activities
Requirements	Find actors and use cases, prioritize use cases, detail use cases, prototype user interface, structure the use case model
Analysis	Architectural analysis, analyze use cases, explore classes, find packages
Design	Architectural design, trace use cases, refine and design classes, design packages
Implementation	Architectural implementation, implement classes and interfaces, implement subsystems, perform unit testing, integrate systems
Test	Plan and design tests, implement tests, perform integration and system tests, evaluate tests

# Workflows and Models

---



# Summary 1

---

A software development process is a framework that provides guidance to carry out the different activities needed to produce a software product.

Every software development process must be parameterized for each individual project.

There is no universal process.

# Summary 2

---

The main features of the Unified Process are:

- 1) use case-driven
- 2) architecture-centric
- 3) iterative and incremental

With respect to time dimension, the lifecycle is divided into phases and iteration.

There are four main phases: inception, elaboration, construction and transition.

A specific set of artifacts are produced with well-defined activities called workflows.

There are five main workflows: requirements, analysis, design, implementation and test.

# Tools

# The Course: Overview

---

- |                             |                         |
|-----------------------------|-------------------------|
| 1) The Course               | 7) Design Model         |
| 2) Object Oriented Concepts | 8) Implementation Model |
| 3) UML Basics               | 9) Deployment Model     |
| 4) Case Study               | 10) UML and UP          |
| 5) Requirement Model        | 11) Tools               |
| 6) Architecture Model       | 12) Summary             |

# Overview

---

- 1) Why UML CASE tools?
- 2) Benefits
- 3) Different tools
- 4) Evaluating UML CASE tools
- 5) Main Features of:
  - a) Enterprise Architect
  - b) MagicDraw
  - c) Poseidon
  - d) Rational Rose

# Why UML CASE tools?

- 1) enable to apply an object oriented methodology
- 2) allow to make abstraction of source code
- 3) allow to model the architecture and the design in such a way that are easier to understand and modify
- 4) enable to use models as blueprint for the system
- 5) enable to manage the project with a time dimension and with high level of abstraction



# Benefits of UML CASE Tools

- 1) the use of these tools offer benefits to everyone involved in a project:
  - a) **analysts** can capture requirements with use case model
  - b) **designers** can produce models that capture interactions between objects
  - c) **developers** can quickly turn the model into a working application
- 2) UML case tool, plus a methodology, plus empowered resources enable the development of the right software solution, faster and cheaper

# Different UML CASE Tools

Tools vary with respect to:

- 1) UML modelling capabilities
- 2) project life-cycle support
- 3) forward and reverse engineering
- 4) data modelling
- 5) performance
- 6) price
- 7) supportability
- 8) easy of use
- 9) ...

# Evaluating CASE Tools

Different Criteria for evaluating CASE tools:

- 1) repository support
- 2) round-trip engineering
- 3) HTML documentation
- 4) UML support
- 5) data modelling integration
- 6) versioning
- 7) model navigation
- 8) printing support
- 9) diagrams views
- 10) exporting diagrams
- 11) platform

# Different UML Tools

---

- **Enterprise Architect**
  - organization: Sparx Systems
  - web-site: <http://www.sparxsystems.com.au/>
- **MagicDraw**
  - organization: No Magic Inc.
  - web-site: <http://www.nomagic.com/>
- **Poseidon**
  - organization: Gentleware
  - web-site: <http://www.gentleware.com/>
- **Rational Rose**
  - organization: IBM
  - web-site: <http://www-306.ibm.com/software/rational/>

# Enterprise Architect

---

Criteria	Features
Platform	Windows
UML Compliance	Support for all 13 UML 2.0 diagrams
Usability	Replication capable – Comprehensive and flexible documentation
Development Environment	Multi-user enabled – Allows to replicate and share projects
Outputs & Code Generation	C++, Java, C#, VB, VB.Net, Delphi, PHP – HTML and RTF document generation - Forward and reverse database engineering
Data Repository	Models are stored in a data repository - Checks data integrity in the data repository – Provides a project browser
Other features	Allows scripting to extend functionality – Project estimation tools – User definable patterns

# Magic Draw

---

Criteria	Features
Platform	Any where Java 1.4 is supported
UML Compliance	Support for UML 1.4 notation and semantics
Usability	Replication capable – Customizable views of UML elements – Customizable elements properties
Development Environment	Multi-user enabled – Lock parts of the model to edit – Commit changes – Model versioning and rollback
Outputs & Code Generation	Code generation and reverse engineering to C++, Java, C# - RTF and PDF document generation
Data Repository	Provides a project browser
Other features	Friendly and customizable GUI – Hyperlinks can be added to any model element

# Poseidon

---

Criteria	Features
Platform	Platform independent – Implemented in Java
UML Compliance	Supports all 9 diagrams of UML 1.4
Usability	Replication capable – Internationalization and localization for several languages
Development Environment	Collaborative environment based on client-server architecture – Locking of model parts – Secure transmission of files
Outputs & Code Generation	VB.Net, C#, C++, CORBA IDL, Delphi, Perl, PHP, SQL DDL – Round trip engineering for Java – Diagram export as gif, ps, eps and svg.
Data Repository	Uses MDR (Meta Data Repository) developed by Sun and based on the JMI (Java Metadata Interface) standard
Other features	Allows to import Rational Rose files

# Rational Rose

---

Criteria	Features
Platform	Windows
UML Compliance	Not fully supported UML 1.4
Usability	The add-in feature allows to customize the environment – User configurable support for UML, OMT and Booch 93.
Development Environment	Parallel multi-user development through repository and private support
Outputs & Code Generation	C++, Visual C++, VB6, Java – Documentation generation – Round trip engineering
Data Repository	Maintains consistency between the diagram and the specification, you may change any of them and automatically updates the information.
Other features	Can be integrated with other Rational products such as RequisitePro, Test Manager



# Summary

---

There exist several CASE Tools supporting Object Oriented modelling with UML.

Different criteria should be considered when evaluating a software tool.

Four tools were presented: Enterprise Architect, Magic Draw, Poseidon and Rational Rose.

# Acknowledgements

---

We would like to thank Dr. Tomasz Janowski and all the members of the eMacao team for their valuable comments and help in preparing this material.