# Course Lecture Schedule

Maria
Susan
Sami
Hyrynsalmi

| Date | Topic | Book Chapter(s) |
|---|---|---|
| Wed 8.9. | Course introduction | |
| Tue 14.9. | Introduction to Software Engineering | Chapter 1 |
| Tue 21.9. | Software Processes | Chapter 2 |
| Mon 27.9 | Agile Software Engineering | Chapter 3 |
| Tue 5.10. | Requirements Engineering | Chapter 4 |
| Mon 11.10. | Architectural Design | Chapter 6 |
| **Wed 20.10.** | **Modeling and implementation** | **Chapters 5 & 7** |
| Mon 1.11. | Testing & Quality | Chapters 8 & 24 |
| Mon 8.11. | Software Evolution & Configuration Management | Chapters 9 & 25 |
| Mon 15.11. | Software Project Management | Chapter 22 |
| Mon 22.11. | Software Project Planning | Chapter 23 |
| Mon 29.11. | Global Software Engineering | |
| Wed 8.12. | Software Business | |
| Mon 13.12. | Last topics | |



# Lecture7
# Design and implementation

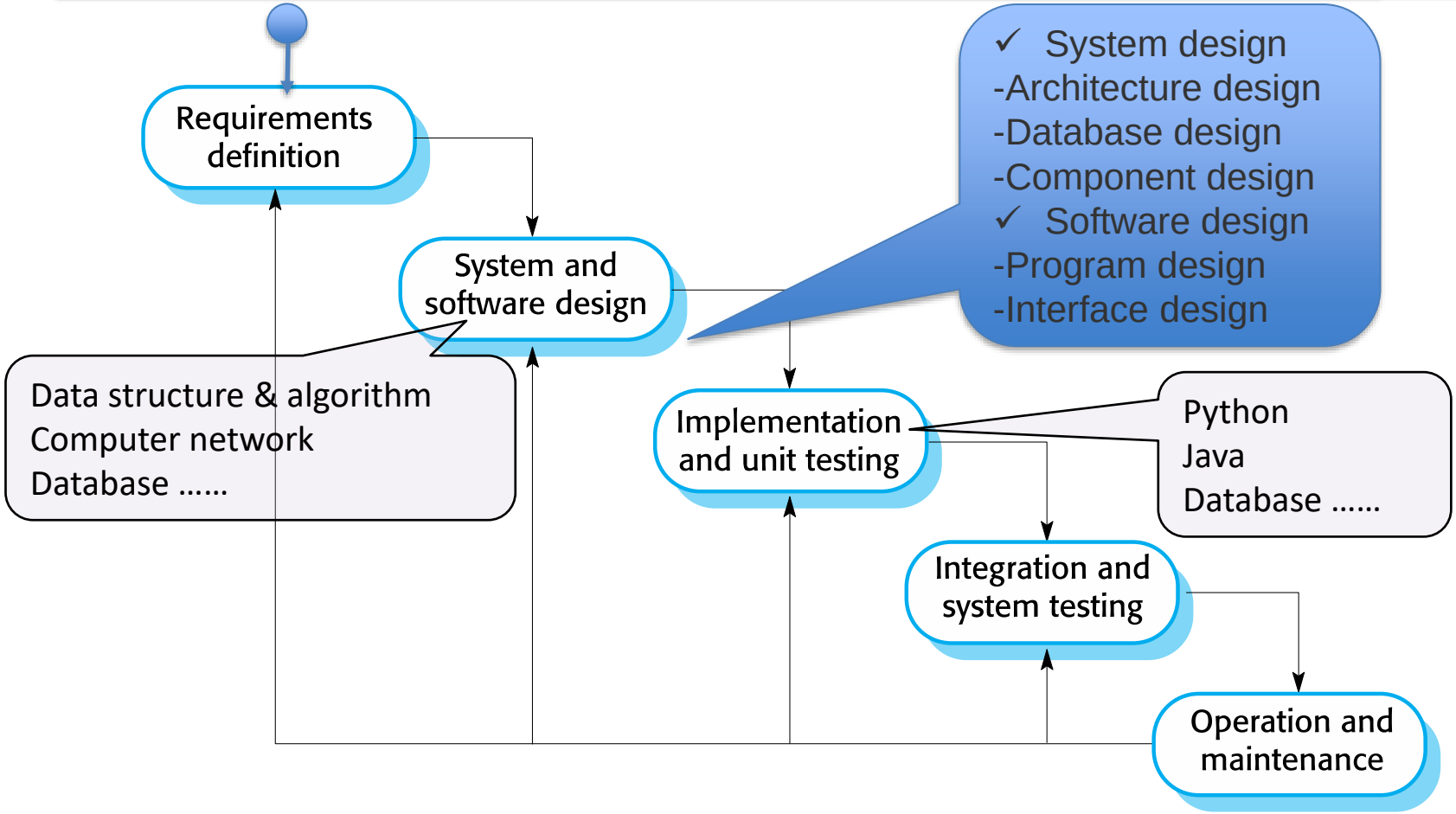
# Topics covered


- ✧ What is design and implementation
- ✧ Object-oriented Software design using the UML
- ✧ Design patterns
- ✧ Implementation issues



# What is design and implementation

# Software design（detailed design）& implementation in waterfall model

# Design and implementation


- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
  - Software design is a creative activity in which you define data-structures and algorithms of software components, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.
- Software design and implementation activities are invariably inter-leaved.

# Structured Programming and object-oriented programming


- Structured programming:
  - In SP, control of program flow is restricted to three structures, sequence, IF THEN ELSE, and DO WHILE, or to a structure derivable from a combination of the basic three.
  - Structured programming language: C, C++, C#, PHP, Ruby, PERL, ALGOL, Pascal, PL/I, and Ada.
- Object oriented programming:
  - OOP is a based on the concept of "objects" and "class", which can contain data (attributes or properties) and code (functions, procedures or methods).
  - Three characteristics of OOP: Encapsulation, inheritance, polymorphism.
  - OOP language: C++, Java, Python, etc.

# Use of graphical models


- Graphical models can be used as a detailed system description.
- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.
- System modeling is now almost always based on notations in the Unified Modeling Language (UML).



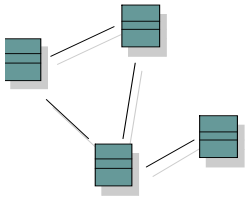
# Object-oriented design using the UML

# Design process stages


- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and the external interactions with the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

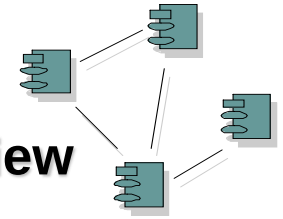
# 4 + 1 view model of software architecture


Modeling user interaction and system-to-system interaction , shows the functional requirements.



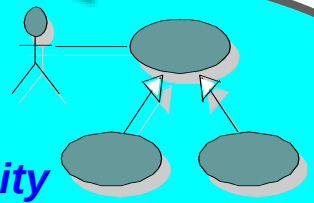
**Logical View**

**Implementation View**



**Analysts/Designers**

***Structure***



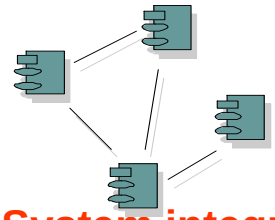
**End-user**

***Functionality***

**Use-Case View**

**Programmers**

***Software management***

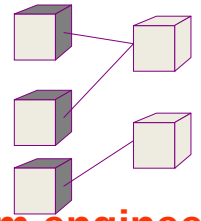


**Process View**

**System integrators**

***Performance***
***Scalability***
***Throughput***

**Deployment View**



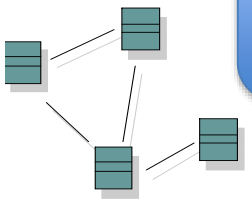
**System engineering**

***System topology***
***Delivery, installation***
***communication***

Modeling the physical structure of the system, shows how software components are distributed across the processors in the system.

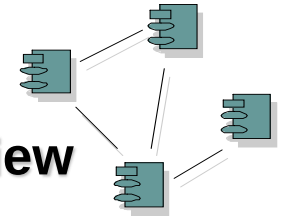
# 4 + 1 view model of software architecture


shows the key abstractions in the system as objects or object classes.



**Logical View**

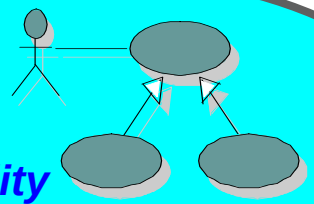
shows how the software is decomposed for development.



**Implementation View**

**Analysts/Designers**

***Structure***



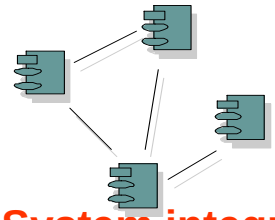
**End-user**

***Functionality***

**Use-Case View**

**Programmers**

***Software management***

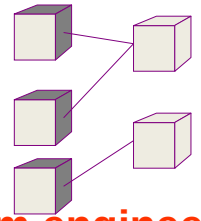


**Process View**

**System integrators**

***Performance***
***Scalability***
***Throughput***

shows at run-time, how the system is composed of interacting processes.

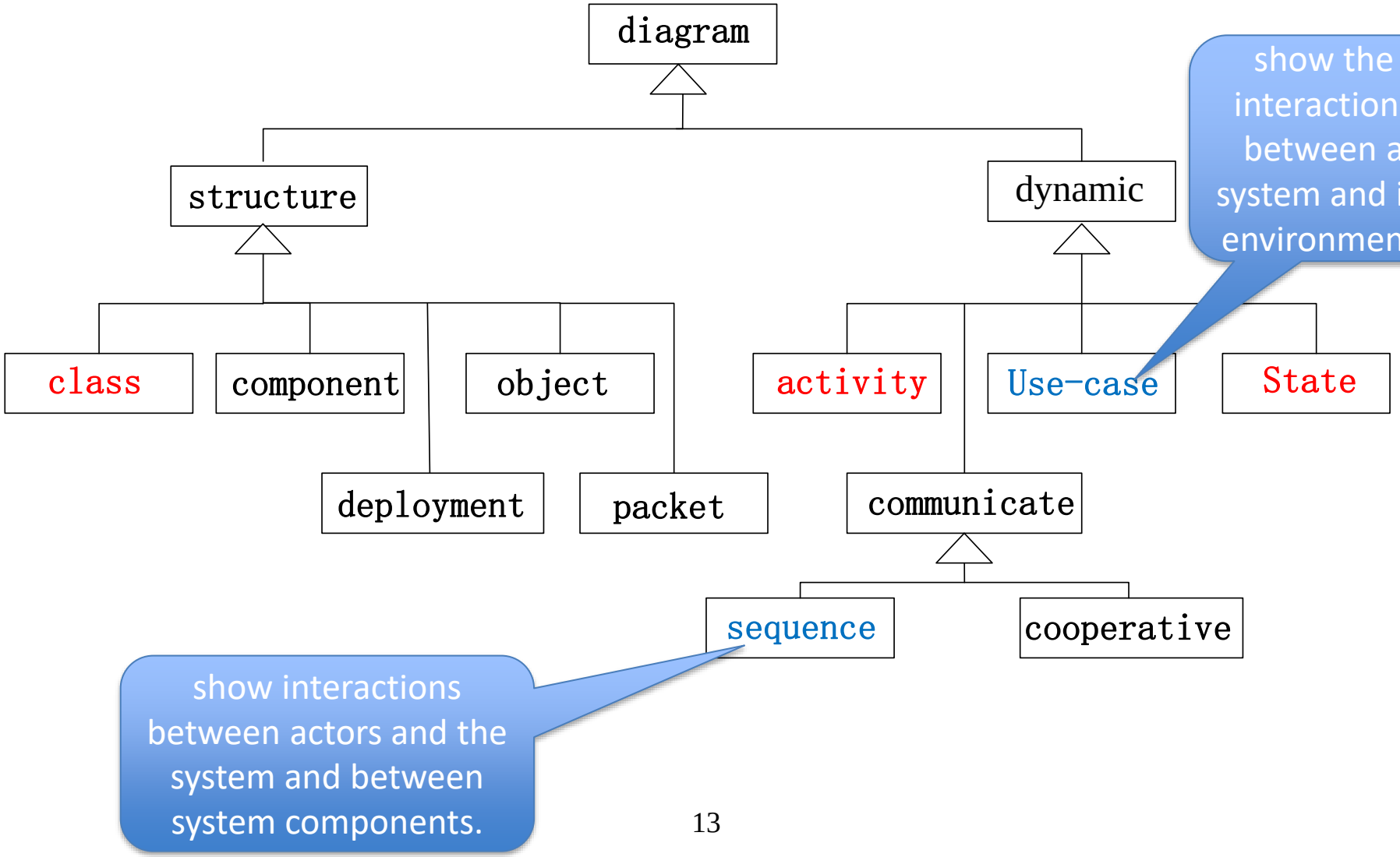


**Deployment View**

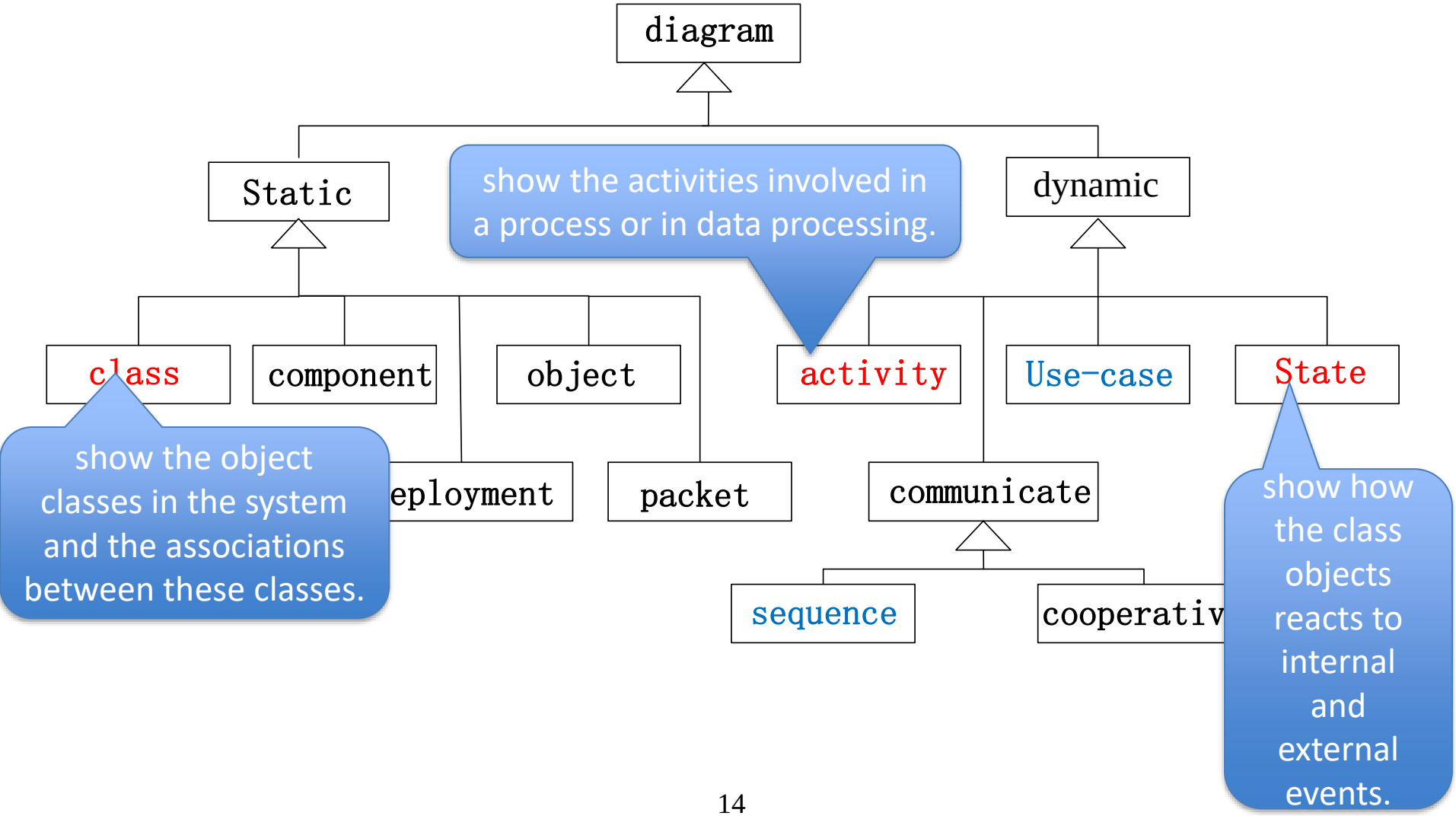
**System engineering**

***System topology***
***Delivery, installation***
***communication***

# UML2.0 Diagrams

# UML2.0 Diagrams

# Class diagrams


- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, a medical department, etc.
- An object class can be thought of as a general definition of one kind of system object.
- Class diagrams are used to show the classes in a system and the associations between these classes.
- https://www.youtube.com/watch?v=Ul6lqHOVHic

# Object class identification


- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

# Approaches to identification


- Use a grammatical approach based on a natural language description of the system.
- Base the identification on tangible things in the application domain.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

# Example: identify the Weather station object classes from the natural language description


Description sentence:

- Weather station use ground thermometer, anemometer, barometer to get original Weather data.

Identify object classes based on noun phrase:

- Weather station use ground thermometer, anemometer, barometer to get original Weather data.

# Example: Weather station object classes


- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

# Weather station object classes


| WeatherStation |
| --- |
| identifier |
| reportWeather ( )<br>reportStatus ( )<br>powerSave (instruments)<br>remoteControl (commands)<br>reconfigure (commands)<br>restart (instruments)<br>shutdown (instruments) |

| WeatherData |
| --- |
| airTemperatures<br>groundTemperatures<br>windSpeeds<br>windDirections<br>pressures<br>rainfall |
| collect ( )<br>summarize ( ) |

| Ground thermometer |
| --- |
| gt_Ident<br>temperature |
| get ( )<br>test ( ) |

| Anemometer |
| --- |
| an_Ident<br>windSpeed<br>windDirection |
| get ( )<br>test ( ) |

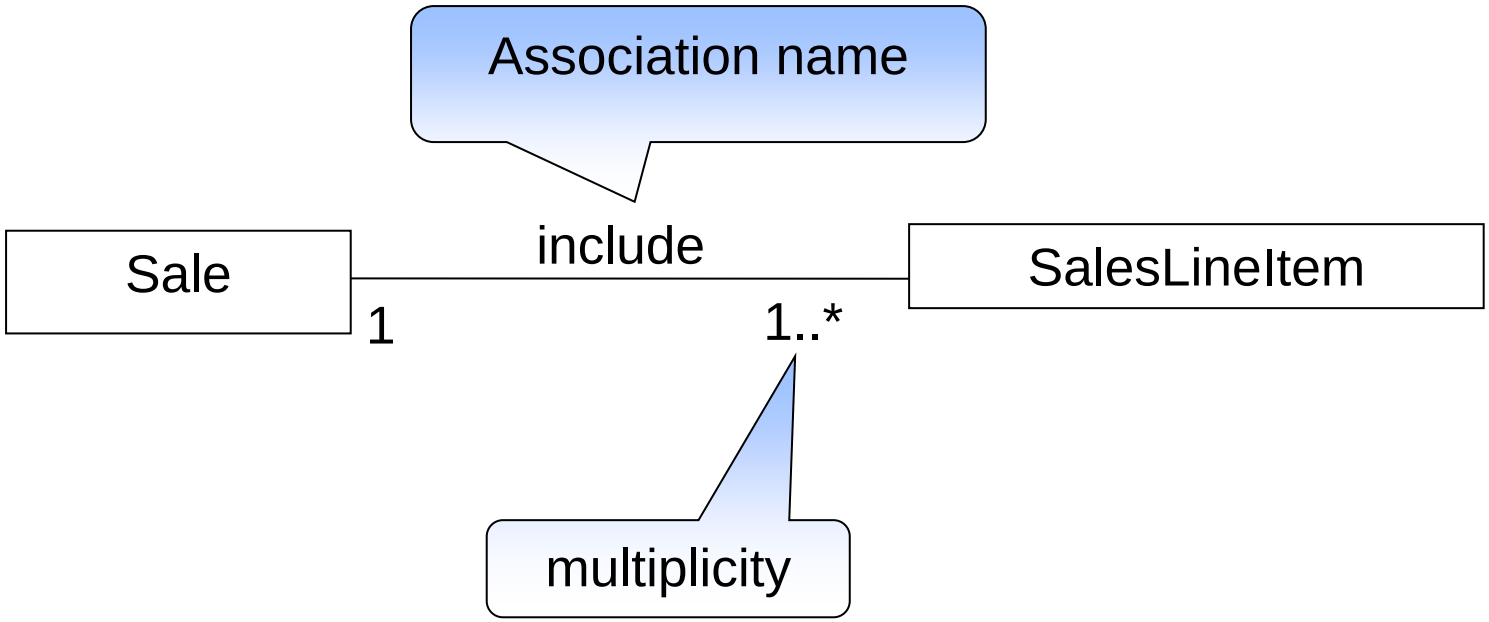
| Barometer |
| --- |
| bar_Ident<br>pressure<br>height |
| get ( )<br>test ( ) |

# Design models


- Design models show the object classes and relationships between these entities.
- There are two kinds of design model:
  - Structural models describe the static structure of the system in terms of object classes and relationships(association, generalization, aggregation).
  - Dynamic models describe the dynamic interactions between objects.

# Object class associations

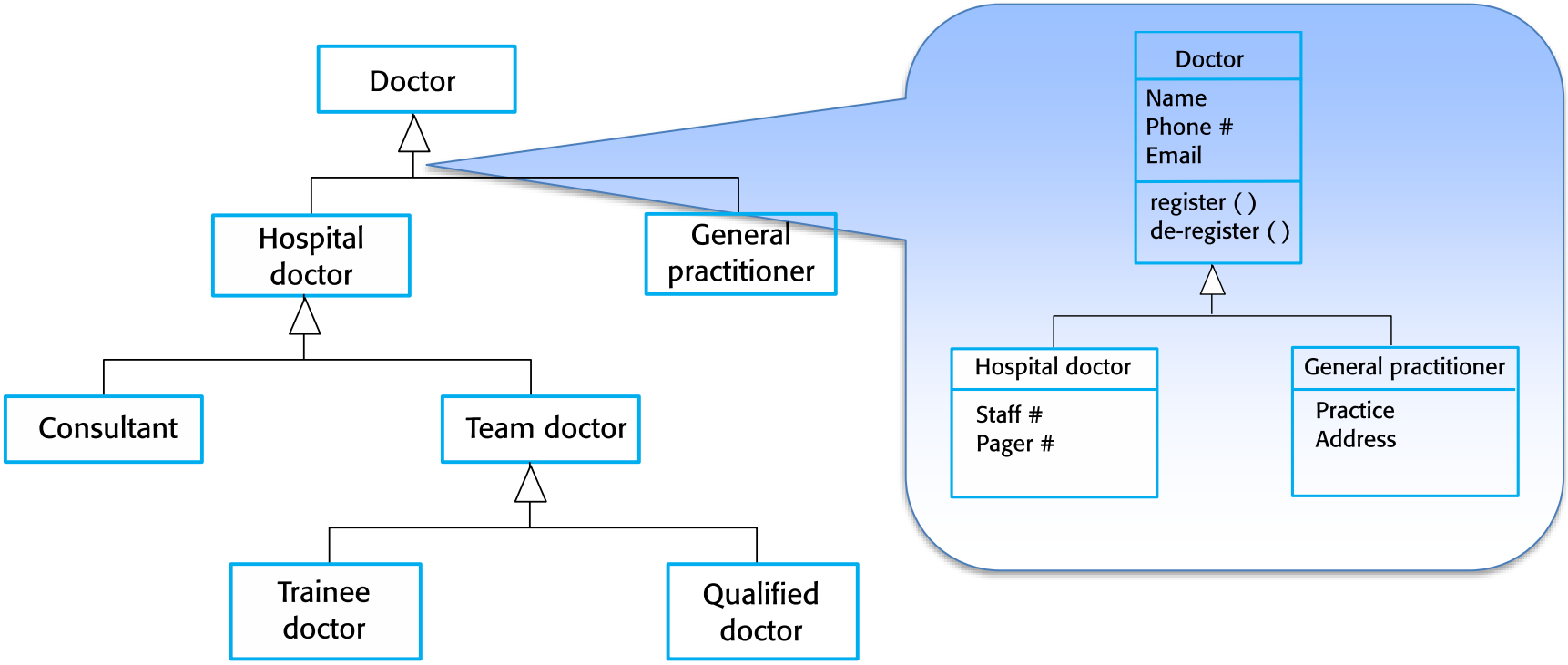

- Association represents a structural relationship between different objects classes.



# Generalization

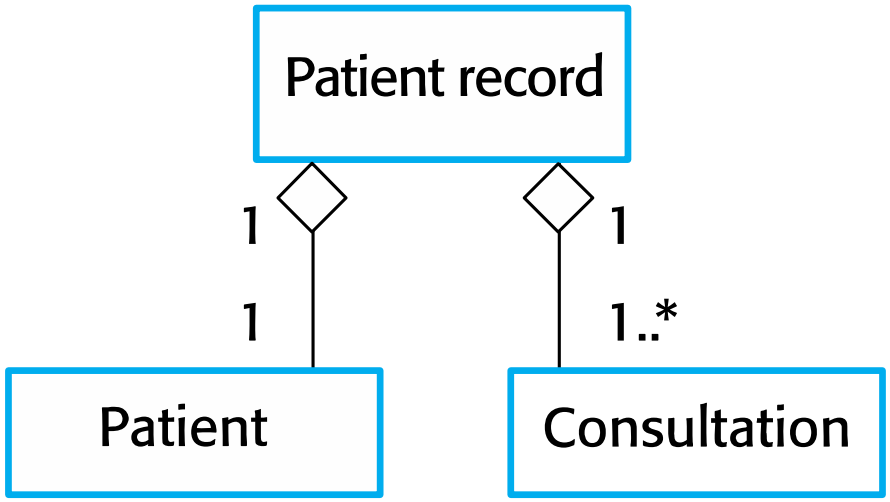

- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.
- In a generalization, the lower-level classes are subclasses inherit the attributes and operations from their super-classes. These lower-level classes then add more specific attributes and operations.
- Attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

# Example: A generalization hierarchy of doctors

# Object class aggregation


- An aggregation shows how collection classes are composed of other classes.
- Aggregations are similar to the part-of relationship in semantic data models.



# Example: class diagram(packet diagram) of vehicle rent company


RentalCompany

Employee
+ id: UnlimitedNatural [1]
+ since: Date [1]

[*] + employee + employer [1]
[1] + consultant + company [1]

VehicleRental
+ name: String [1]
+ headquarters: Address [1]
+ services: VehicleRental [*]

[1] + owner [*] + truck
+ owner [1]

Truck
+ weight: Real [1]
+ power: Integer [1]
+ manufacturer: String [1]
+ regnumber: String [1]

+ customer [*] [*] + customer

Customer
+ id: UnlimitedNatural [1]
+ since: Date [1]
+ account: BankAccount [1]

[1] + owner
+ motorbike [*]
+ car [*]

Motorbike
+ power: Integer [1]
+ manufacturer: String [1]
+ cylindercap: UnlimitedNatural [1]
+ type: MotorbikeType [1]
+ regnumber: String [1]

Car
+ typ: CarType [1]
+ power: Integer [1]
+ regnumber: String [1]
+ seats: UnlimitedNatural [1]
+ manufacturer: String [1]

Subcontractor
+ office: Address [1]

Invoicing

BankAccount
+ IBANcode: String [1]
+ depositor: Person [1]
+ bankname: String [1]

Invoice
+ customer: Customer [1]
+ amount: UnlimitedNatural [1]
+ rental: VehicleRental [1]
+ paid: Boolean [1]

Commonalities

*Person*
+ forename: String [1]
+ surname: String [1]
+ email: String [0..1]

[*] + person
[1] + address

Address
+ street: String [1]
+ number: UnlimitedNatural [1]
+ postalcode: String [1]
+ city: String [1]

Date
+ day: UnlimitedNatural [1]
+ month: UnlimitedNatural [1]
+ year: UnlimitedNatural [1]
+ isBefore( + d: Date): Boolean

«Enumeration» CarType
LIMOUSINE
CABRIOLET

«Enumeration» MotorbikeType
CHOPPER
CRUISER

*Vehicle*
+ rentable: Boolean [1]
+ rent()

realizes

«Interface» Rentable
+ rent()

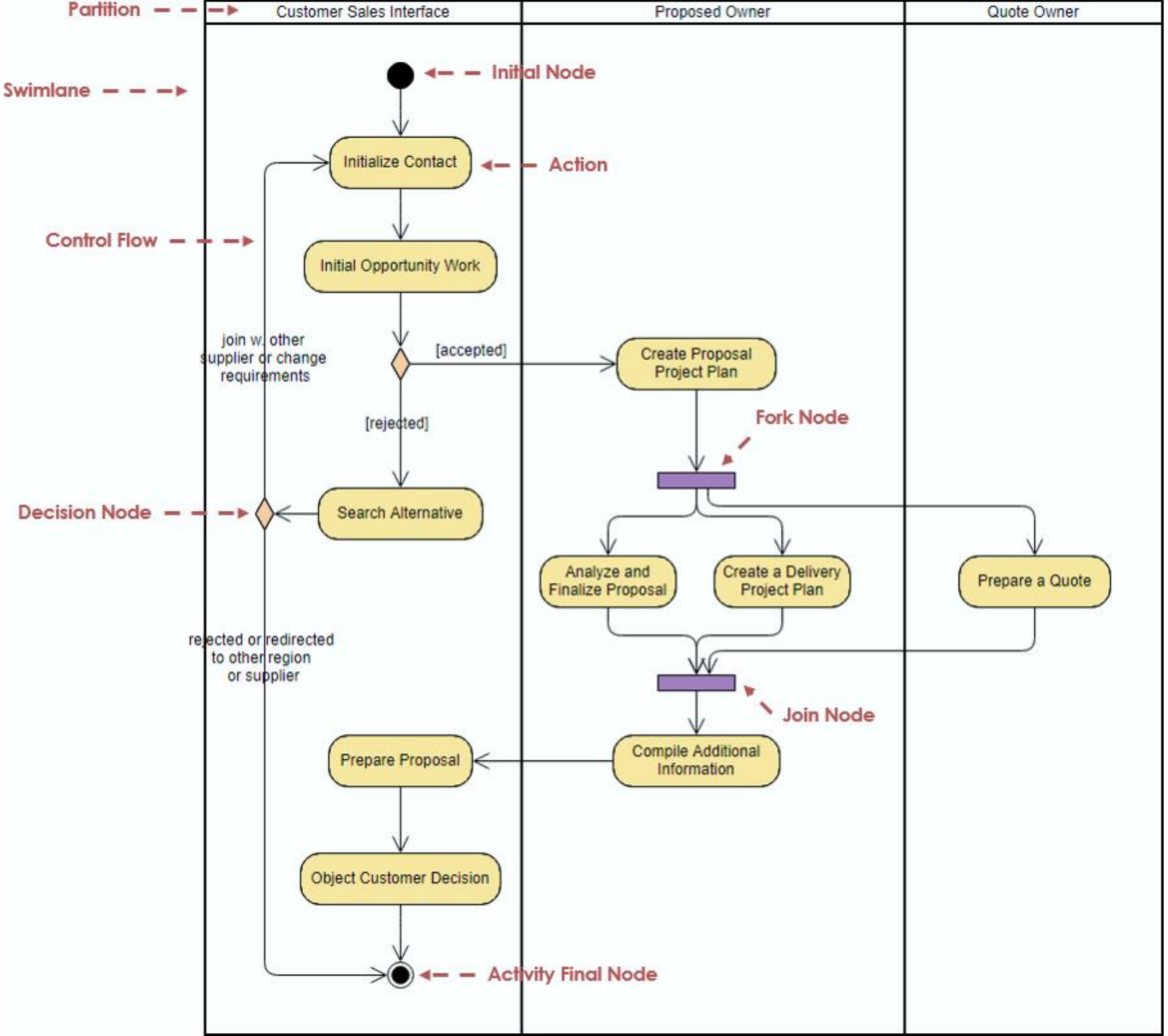
Services

*Service*
+ customer: Customer [1]
+ vehicle: Vehicle [1]

VehicleRental
+ from: Date [1]
+ to: Date [1]
+ rentCar( + car: Car)
+ rentTruck( + truck: Truck)
+ rentMotorbike( + mb: Motorbike)

# Dynamic models


- Dynamic models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.


Partition
Customer Sales Interface
Proposed Owner
Quote Owner
Swimlane
Initial Node
Initialize Contact
Action
Control Flow
Initial Opportunity Work
join w. other supplier or change requirements
[accepted]
Create Proposal Project Plan
[rejected]
Fork Node
Decision Node
Search Alternative
Analyze and Finalize Proposal
Create a Delivery Project Plan
Prepare a Quote
rejected or redirected to other region or supplier
Join Node
Prepare Proposal
Compile Additional Information
Object Customer Decision
Activity Final Node

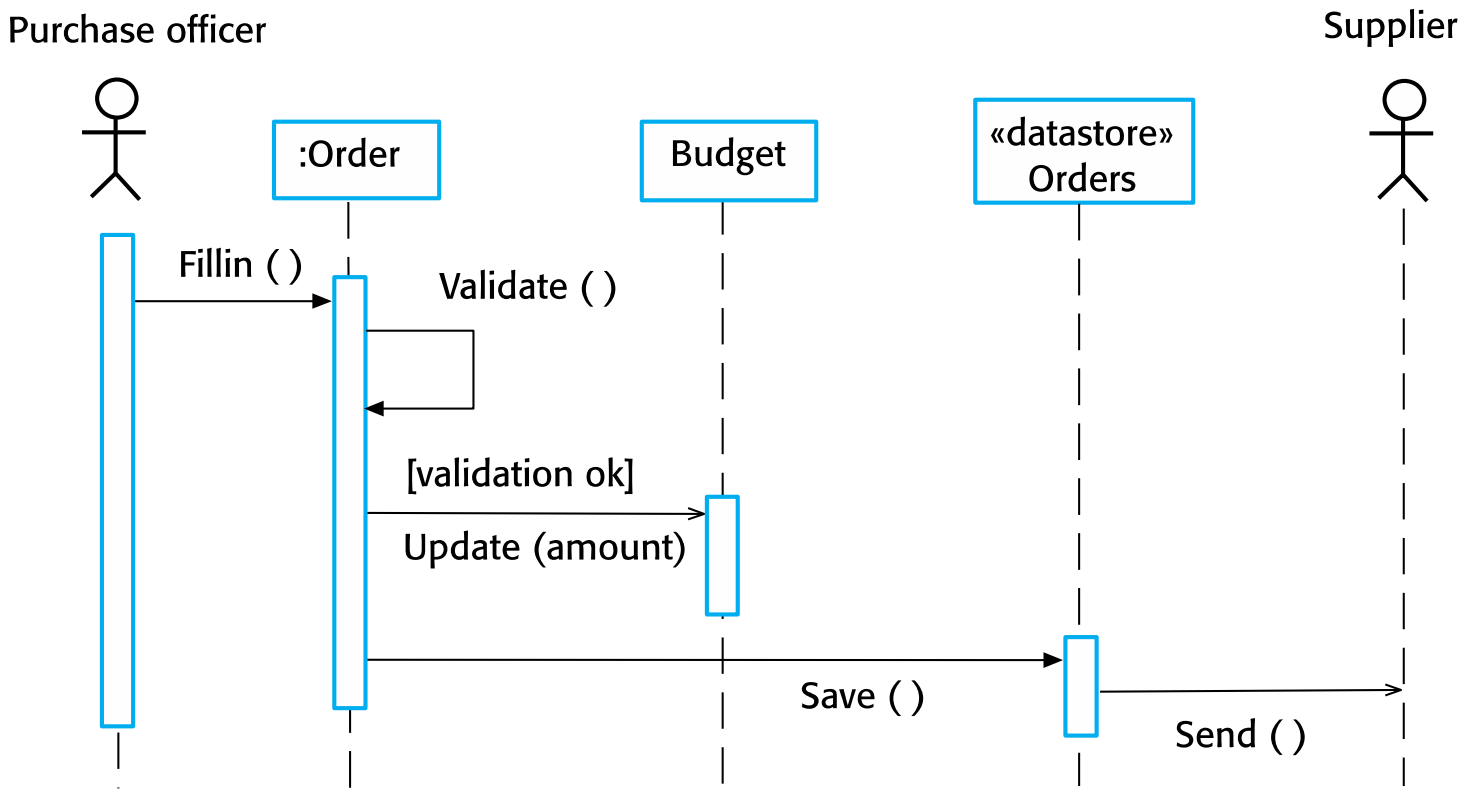
# Behavioral models


- Behavioral models are models of the dynamic behavior of a system responds to a stimulus from its environment.
- The stimulus from its environment being of two types:
  - Data - Some data arrives that has to be processed by the system.
  - Events - Some event happens that triggers system processing.

# Data-driven modeling


- Many business systems are data-processing systems that are primarily driven by data.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

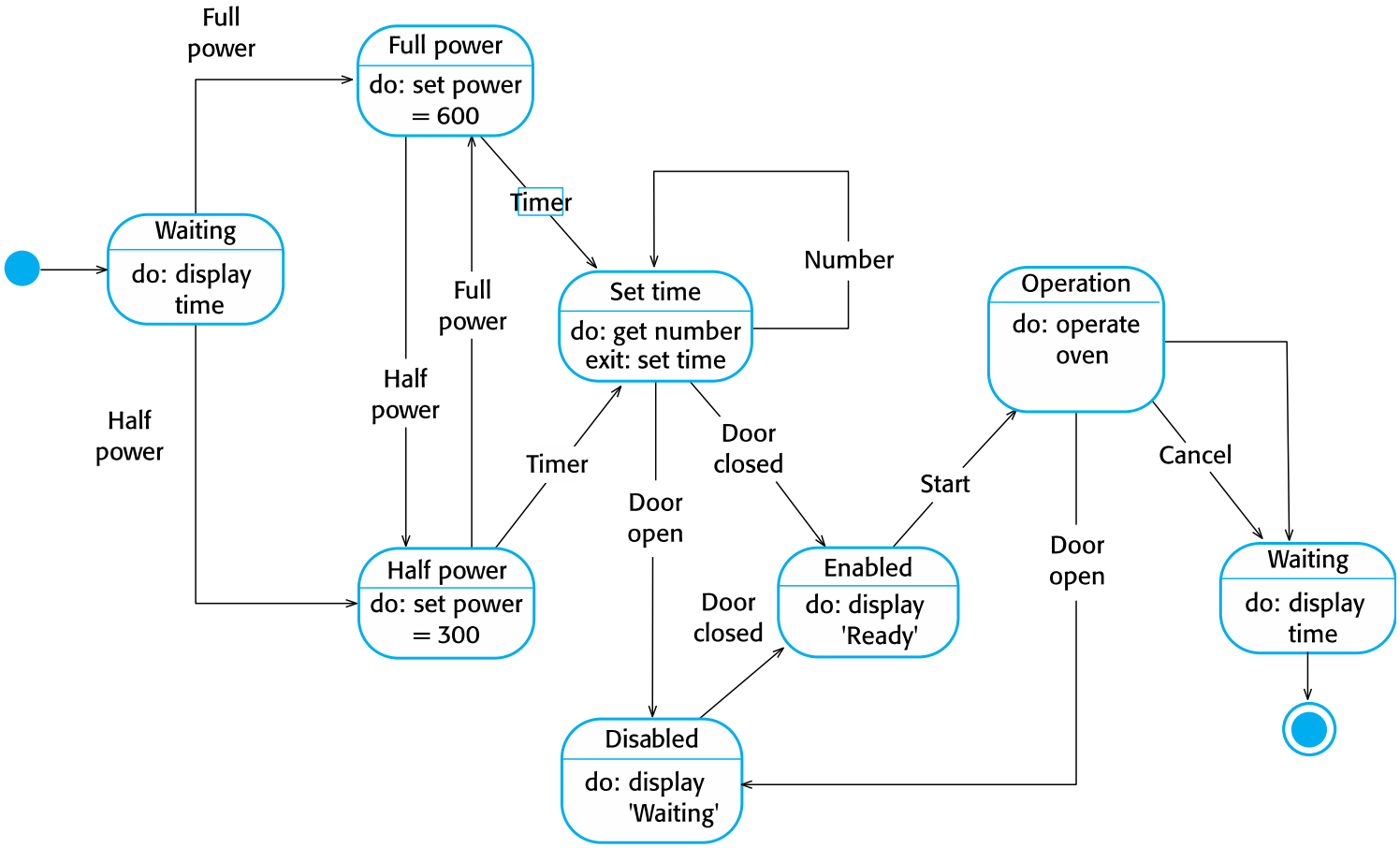
# Example: Data-driven order processing

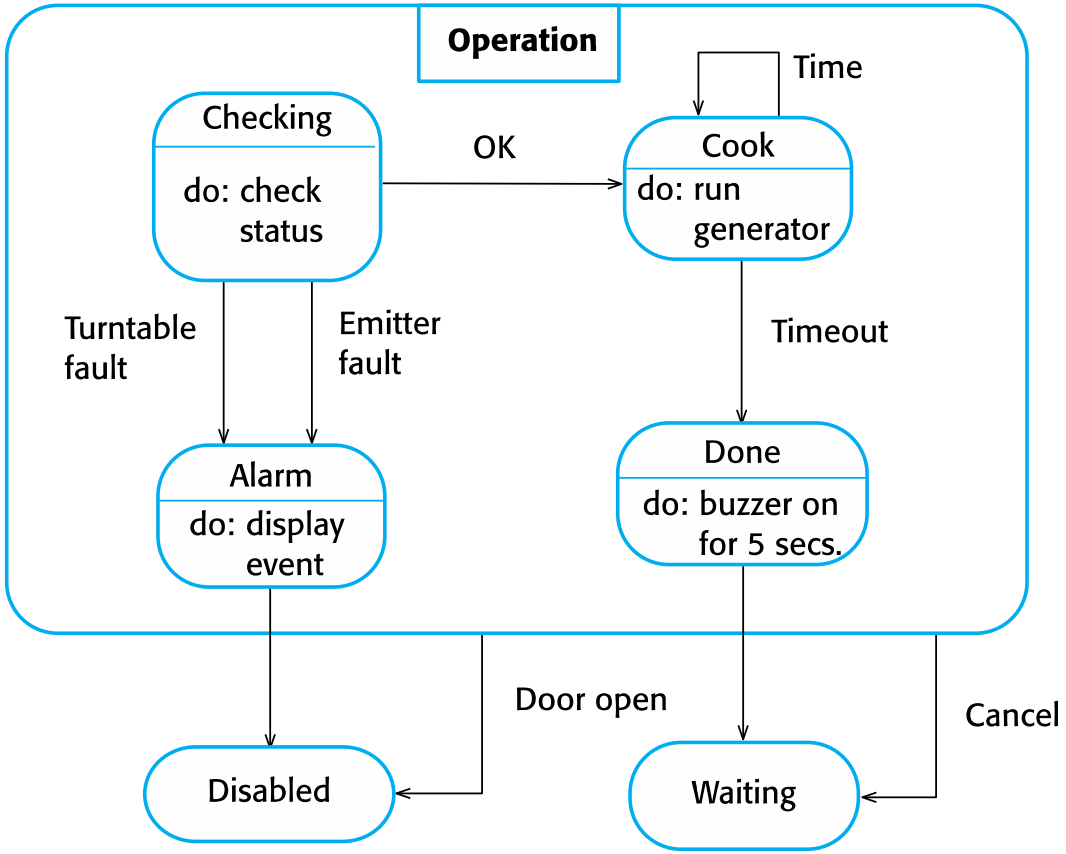
# Event-driven modeling


- Real-time systems are often event-driven.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.

# Example: State diagram of a microwave oven

# Microwave oven operation



# States and stimuli for the microwave oven (a)


| State | Description |
|---|---|
| Waiting | The oven is waiting for input. The display shows the current time. |
| Half power | The oven power is set to 300 watts. The display shows ‘Half power’. |
| Full power | The oven power is set to 600 watts. The display shows ‘Full power’. |
| Set time | The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set. |
| Disabled | Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’. |
| Enabled | Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’. |
| Operation | Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding. |

# States and stimuli for the microwave oven (b)


| Stimulus | Description |
|---|---|
| Half power | The user has pressed the half-power button. |
| Full power | The user has pressed the full-power button. |
| Timer | The user has pressed one of the timer buttons. |
| Number | The user has pressed a numeric key. |
| Door open | The oven door switch is not closed. |
| Door closed | The oven door switch is closed. |
| Start | The user has pressed the Start button. |
| Cancel | The user has pressed the Cancel button. |

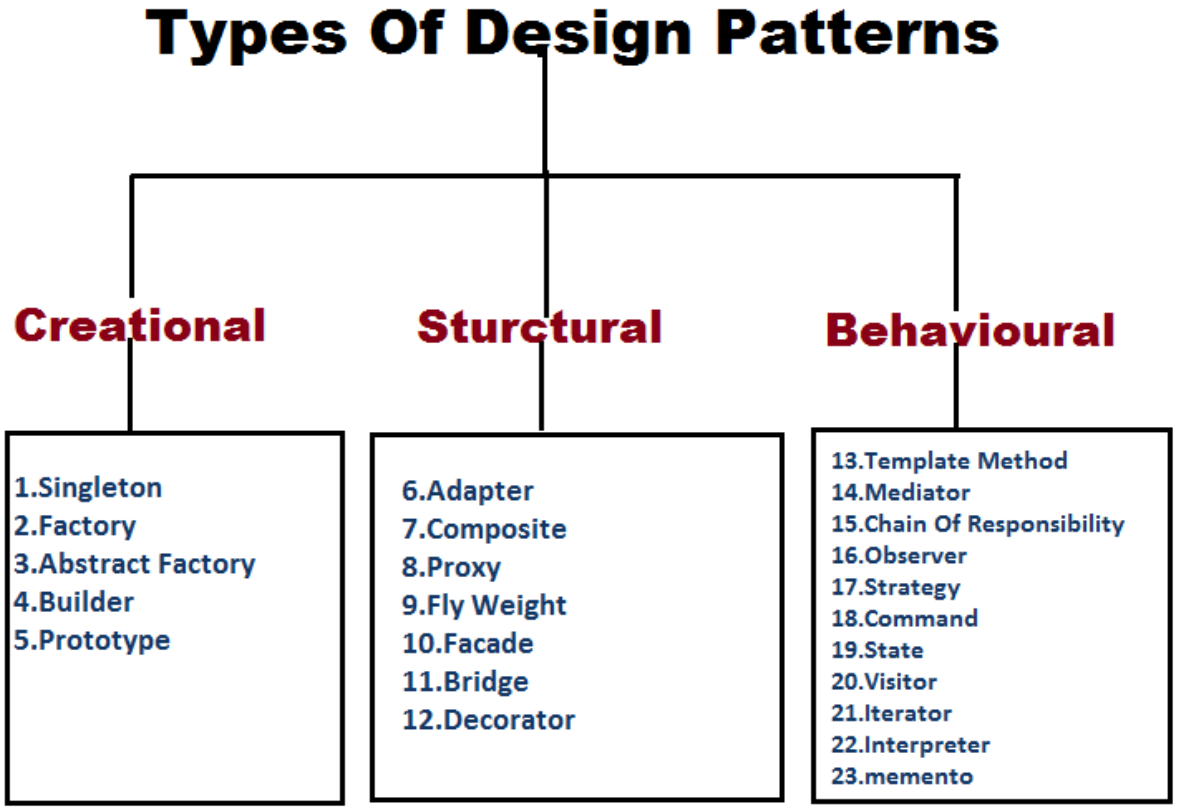


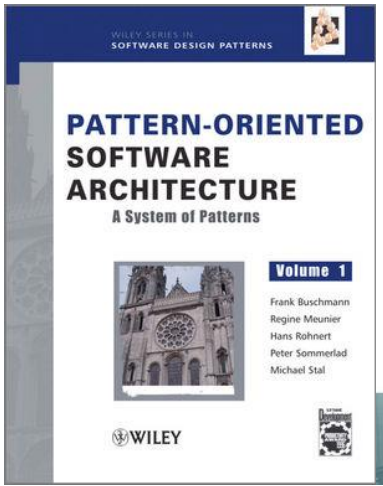
# Design patterns

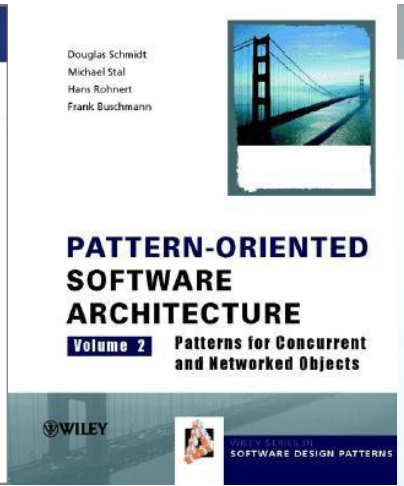
# Design patterns

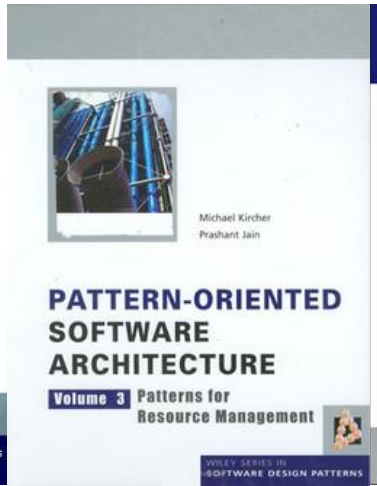

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

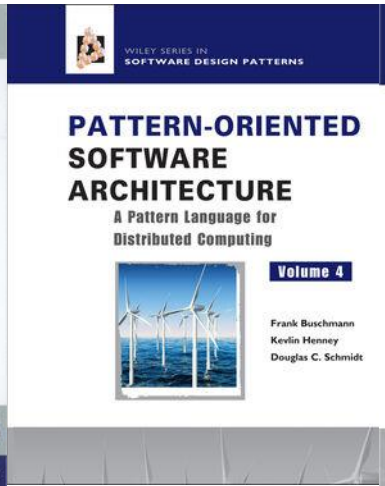
# Pattern-Oriented Software Architecture Series

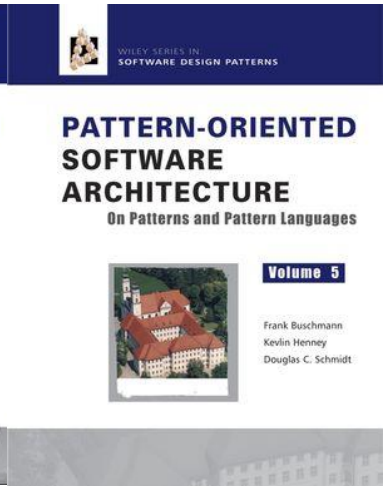


# Pattern elements


- ✧ Name
  - A meaningful pattern identifier.
- ✧ Problem description.
- ✧ Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- ✧ Consequences
  - The results and trade-offs of applying the pattern.

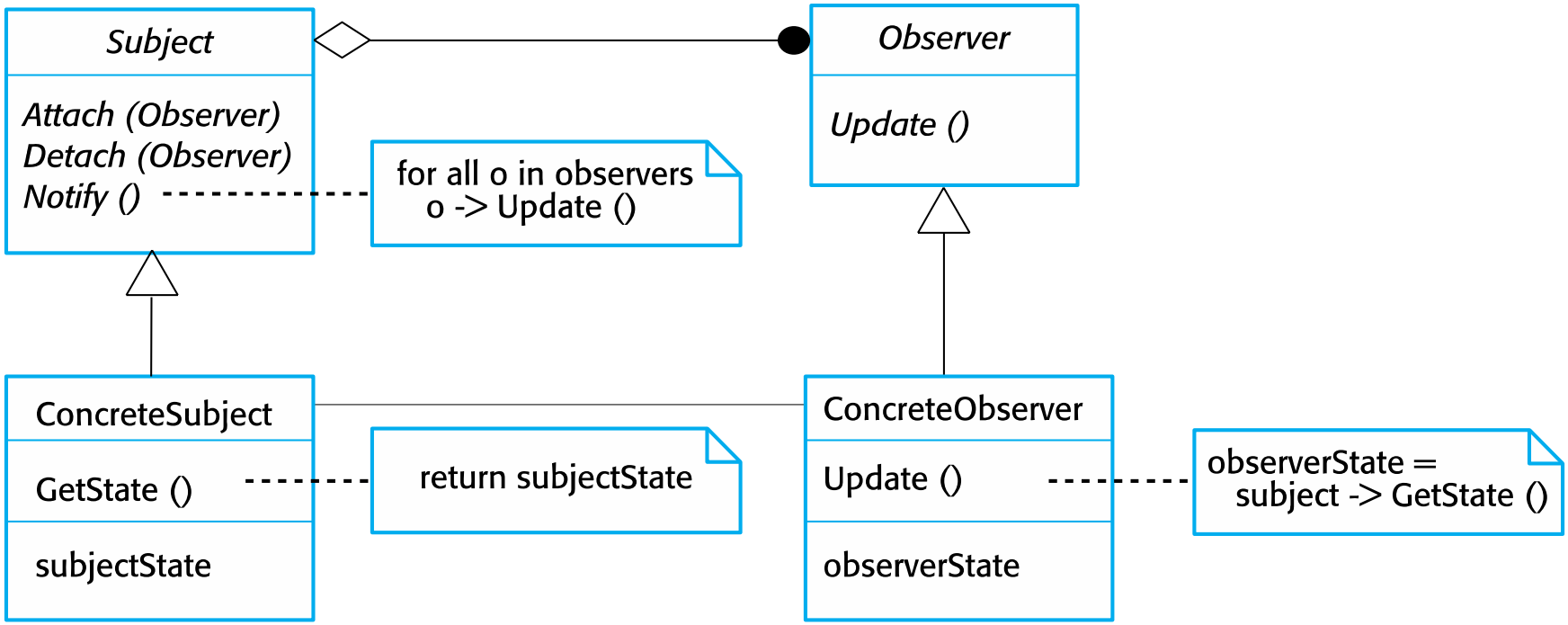
# The Observer pattern (A)


| Pattern name | Observer |
|---|---|
| Description | Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change. |
| Problem description | In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.<br>This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used. |

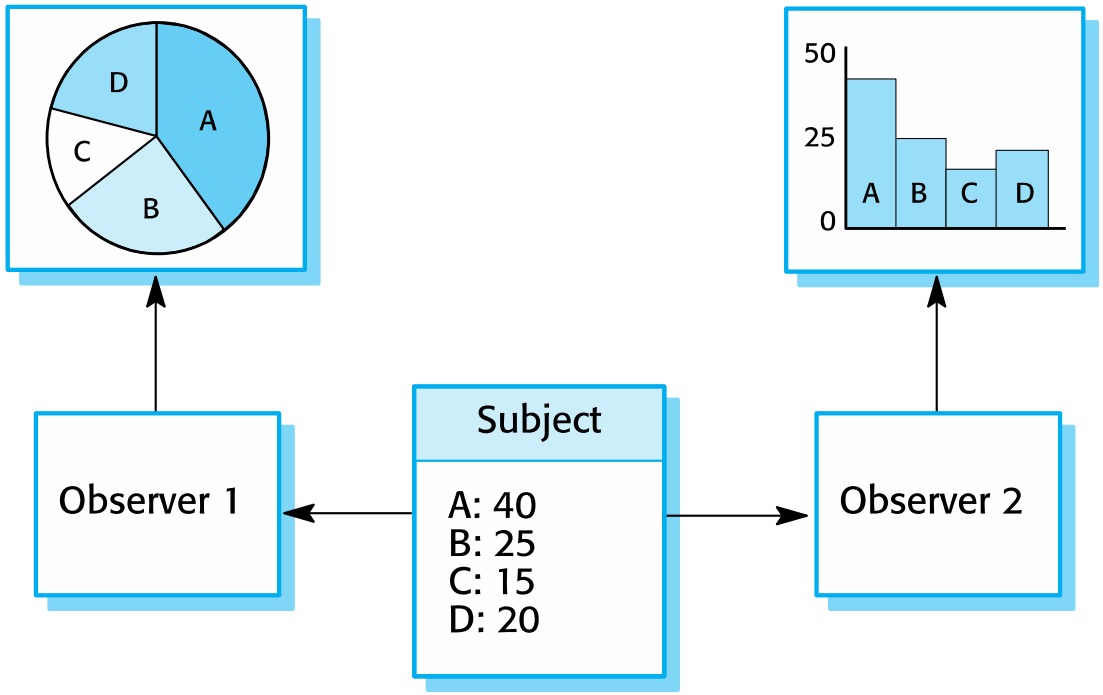
# The Observer pattern (B)


| Pattern name | Observer |
|---|---|
| Solution description | This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.<br><br>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated. |
| Consequences | The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary. |

# A UML model of the Observer pattern

# Example: Multiple displays using the Observer pattern



# Implementation issues

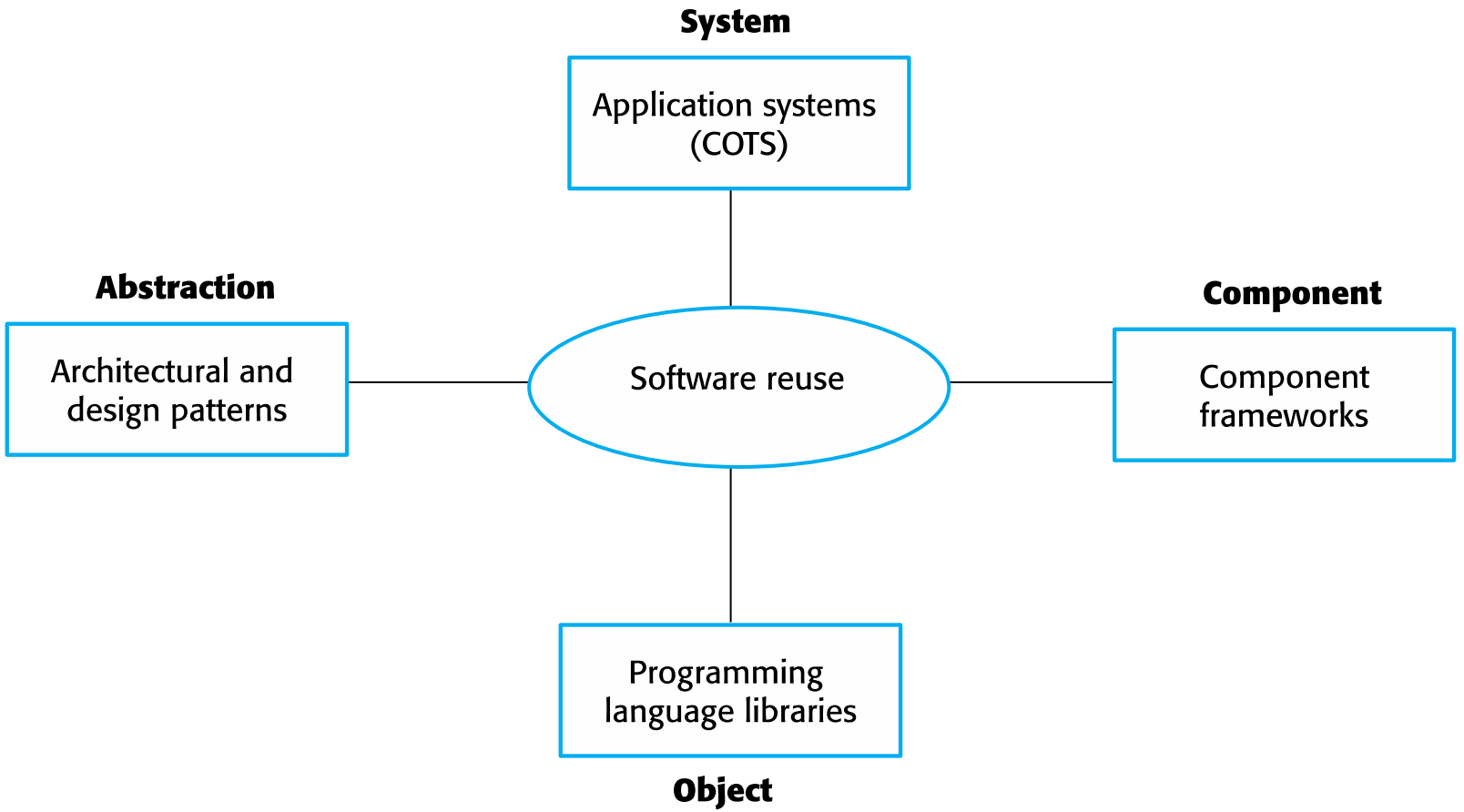
# Implementation issues


- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse: Most modern software is constructed by reusing existing components or systems.
  - Configuration management: During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Host-target development: Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

# Reuse


- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based on the reuse of existing software emerged and is now generally used for business and scientific software.

# Software reuse levels

# Reuse costs


- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- The costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

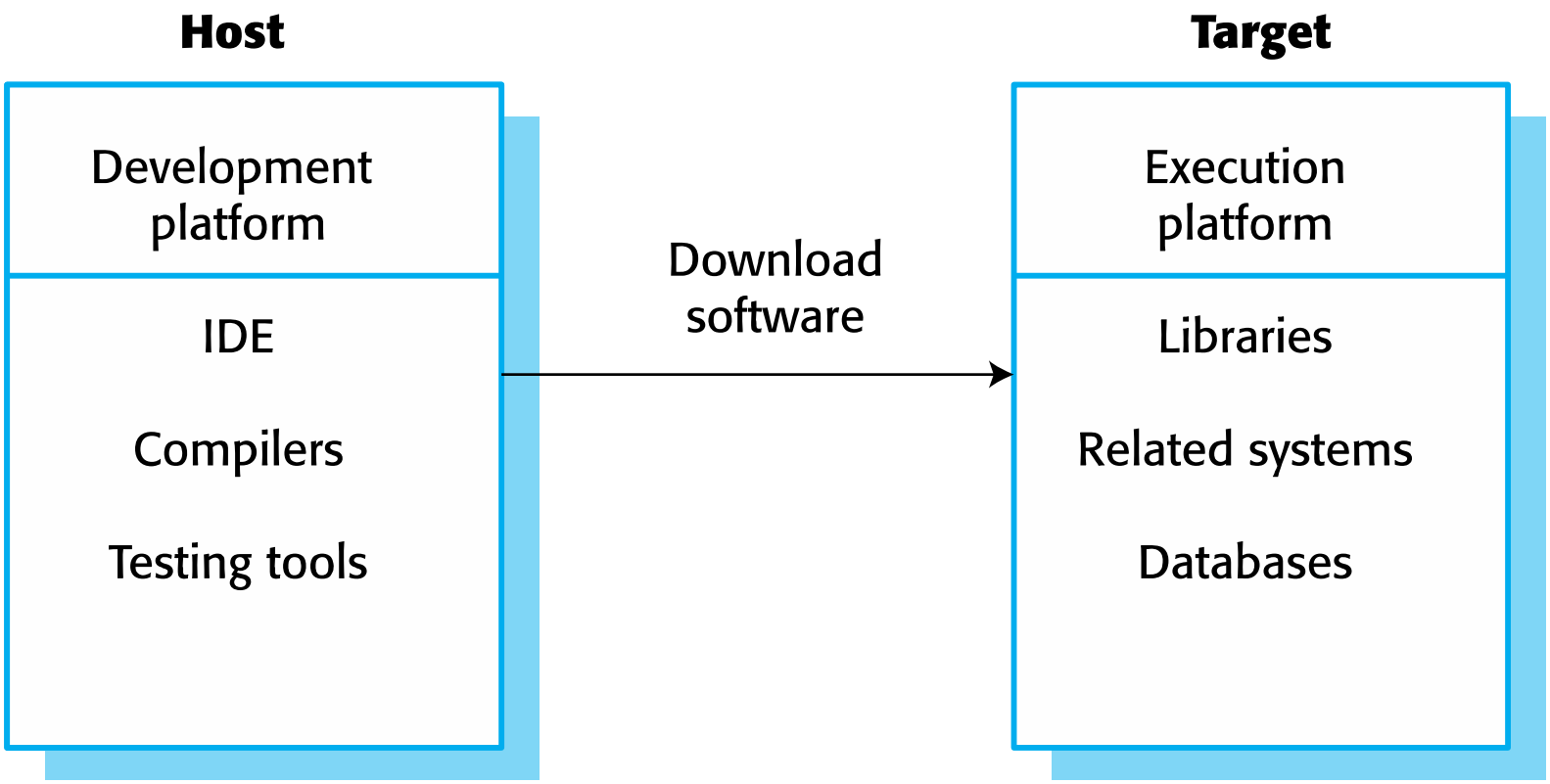
# Configuration management


- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- The details will be talked about in lecture9.

# Host-target development


- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

# Host-target development

# Component/system deployment factors


- The hardware and software requirements: If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- The availability requirements: High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- Component communications: If there is a high level of communications traffic between components, it usually require to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

# Development platform tools


- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Tools to support refactoring and program visualization.
- Configuration management tools to manage source code versions and to integrate and build systems.

# Lecture 7:


**Reading for this week:**

-Chapter 5：System Modeling from the coursebook (Software Engineering by Ian Sommerville)

- Chapter 7: Design and Implementation from the coursebook (Software Engineering by Ian Sommerville)

**Watch the videos:**

1. https://www.youtube.com/watch?v=3cmzqZzwNDM (UML 2.0 Class Diagrams,17 min)
2. https://www.youtube.com/watch?v=XFTAIj2N2Lc (UML 2.0 Activity Diagrams ,12 min)
3. https://www.youtube.com/watch?v=cxG-qWthxt4 (UML 2.0 Sequence Diagrams,12 min)
4. https://www.youtube.com/watch?v=_6TFVzBW7oo (UML 2.0 State Machine Diagrams,13 min)

# Assignments for this week:


- ✧ the assignment is not a typical essay, but instead of an essay all will do this same modeling assignment.
- ✧ no Turnitin check is needed this week, but cooperation is strictly forbidden and all need to do this assignment on his/her own.

1. Explain the following concepts, as they relate to software modeling. Use only a couple of sentences for each answer.

2. Look carefully at how messages and mailboxes are represented in the email system that you use.

Model the object classes that might be used in the system implementation to represent a mailbox and an e-mail message. Include attributes, methods and relationships.

Document your model as a UML class diagram.

# Other things for attention:


- The project will include modeling. The project will be done in teams that Marianne will create. We will inform about the details later on.
- Lecture 8-9 will be given online.