REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
الجمهورية الجزائرية الديمقراطية الشعبية

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

HIGHER SCHOOL IN APPLIED SCIENCES
--T L E M C E N--

وزارة التعليم العالي والبحث العلمي

المدرسة العليا في العلوم التطبيقية
-تلمسان-

Graduate Studies Department

Course Handout

Field of study: Industrial Engineering
Specialization: Industrial Management and Logistics

Presented by: Kouloughli Imane
Associate Professor, class A

Title

Course Handout « Industrial Information Systems »

Academic Year: 2025/2026

Preface

This course handout is intended for students in the **second cycle, Industrial Engineering specialization**. It is part of the course titled **"Industrial Information Systems"**, a foundational subject for understanding the challenges related to the management, control, and digital transformation of modern production systems.

The course is organized into three main chapters. The first introduces the general concepts of information systems, explaining their role, components, and growing importance in industrial organizations. The second chapter focuses on **ERP (Enterprise Resource Planning)** systems, with particular emphasis on **Odoo**, an open-source software increasingly used in industry for business process integration. The third chapter covers the principles of **modeling with UML (Unified Modeling Language)**, a key tool for designing and representing information systems.

The aim of this handout is to provide students with both the theoretical foundation and practical tools needed to understand, analyze, and model information systems in an industrial context. It also seeks to raise awareness of the challenges of Industry 4.0, where information is a strategic driver of performance and innovation.

I hope this material will support your learning journey and spark your interest in this discipline at the heart of industrial digitalization.

Table of Contents

Chapter 1: Industrial Information Systems.....7

1. Introduction.....7
2. The different flows.....7
   - 2.1 Physical flows.....7
   - *2.2 Monetary flows*.....8
   - *2.3 Information flows*.....8
2. The different subsystems within an organization:.....8
3. Information system (IS).....10
   - *3.1 The functions of an information system*.....11
   - *3.2 Qualities of an Information System*.....12
   - *3.3 Examples of Information Systems*.....13
   - *3.4 Main Needs of Information Systems Integration (SII):*.....17
4. Conclusion.....18

Chapter 2: ERP (Enterprise Resource Planning) Systems.....20

1. Introduction.....20
2. What is an ERP?.....20
3. Principle of ERP Systems.....21
4. ERP Functionalities.....22
5. Key Characteristics of an ERP System.....22
   - *5.1 Use of a Single Database: Historical Evolution of Management Information Systems*.....23
   - *5.2 Integration into an ERP System*.....24
   - *5.3 ERP Modules Cover All Business Activities*.....25
   - *5.4 High Configurability*.....25
   - *5.5 Decision Support Tool*.....26
6. Types of ERP Systems.....26
7. The Odoo ERP System.....27
   - *7.1 Odoo Architecture*.....28
   - *7.2 ORM: Object-Relational Mapping*.....29
   - *7.3 Languages and Technologies Used in Odoo*.....30
   - *7.4 Structure of an Odoo Module*.....33

Objectives.....34

Tasks.....35

Odoo Architecture.....35

Structure of an Odoo Module 36
Tasks 50
7.5.3 The Presentation Layer in Odoo – Views, Menus, and Actions 50
Objective 50
Views 50
Tree Views 51
Form Views 52
Below is an example of a Form View display (Figure 36). 53
Actions 53
Menus 54
7.5.4 Inheritance in Odoo 55
Objective 55
View Inheritance 56
Tasks 57
8. Conclusion 58
Chapter 3: Unified Modeling Language (UML) 60
1. Introduction 60
2. Definitions of UML: 60
3. UML Diagrams: 61
4. Use Case Diagram 62
4.1 Main Elements of a Use Case Diagram 63
5. Class Diagram 67
6. Sequence Diagram 82
6.1 Core Elements: 82
Exercises 87
Exercise 1 87
Exercise 2 88
Exercise 3: Production Workshop Management 89
Exercise 4: Inventory Management System 90
Exercise 5: Industrial Maintenance System 91
Exercise 6: Quality Control System 91
Exercise 7: Integrated Industrial ERP System 92
Bonus Reflection 92
Exercise 8: 93

Exercise 9: Production Order Management 93
Exercise 10: Inventory Management 94
Exercise 11: Industrial Maintenance 94
Exercise 12: Purchasing Management (Advanced Level) 95
Exercise 13: Mini Industrial ERP System 95
Exercise 14: 96
Mini Project: UML Modeling and Odoo Module Development "School Library Management at ESSAT" 96
Learning Objectives 96
Project Context 97
Part 1: UML Analysis and Modeling 97
Requirements Identification 97
Use Case Diagram 97
Class Diagram 98
Sequence Diagram 98
Activity Diagram 98
State Diagram (Optional but Recommended) 99
Part 2: Odoo-Oriented Design 99
Part 3: Odoo Module Development 99
1. Module Creation 99
2. Models (models.py) 99
3. Views (XML) 99
4. Business Logic 99
5. Security 100
Part 4: Testing and Validation 100
Expected Deliverables 100
Constraints and Recommendations 100
7. Conclusion 101
References: 102

List of Figures

Figure 1 Information, money, and material flows in logistics. 8
Figure 2 Different Modules in an ERP System 21
Figure 3 Centralised database architecture *Source*: SOLIDitech. (2021, July 13). *An Introduction to Centralised Databases* [Image]. The SOLID Blog. https://blog.soliditech.com/blog/an-introduction-to-a-centralised-databases....22
Figure 4 Independent Applications 24
Figure 5 Integration into an ERP 24
Figure 6 Distribution of leading ERP vendors in the market 27
Figure 7 The MVC architecture....28
Figure 8 Modular Architecture of Odoo....29
Figure 9 Example of a valid xml file 32
Figure 10 Creating a database with Odoo 36
Figure 11 Structure of an Odoo module....37
Figure 12 Example of a Manifest file 38
Figure 13 Init File 39
Figure 15 Using the scaffold command scaffold command 40
Figure 14 Views File 40
Figure 16 Module added to the addons directory 41
Figure 17 Odoo Main Interface 42
Figure 18 Restarting the Odoo server 42
Figure 19 Accessing Developer Mode 43
Figure 21 Activating Developer Mode....43
Figure 20 Updating the applications list 43
Figure 22 Display of the created module in the Odoo main interface 44
Figure 23 Relational database....44
Figure 24 Entering the password 44
Figure 25 Model creation file 46
Figure 26 Loading the student.py file 47
Figure 27 Viewing the created model 47
Figure 28 Relational fields 49
Figure 29 Definition of models with relational fields 49
Figure 30 XML file containing the declaration of a view 50
Figure 32 Tree view 51
Figure 31 Declaration of a view in an XML file 51
Figure 33 Display of a Tree View 52
Figure 34 Display of a List View....52
Figure 35 Form view associated with our model....53
Figure 36 Display of a Form View 53
Figure 37 Example of an action declaration....54
Figure 38 Example of menus and submenus....55
Figure 39 Model inheritance 56
Figure 40 Model inheritance and creation of a new class 56
Figure 41 View inheritance 57
Figure 42 Structure of an Odoo module....58

Figure 43 UML diagrams structural and behavioral 62
Figure 44 Main components of a use case diagram 63
Figure 45 Example of a use case diagram 64
Figure 46 Role generalization 65
Figure 47 Include use case Relationship 66
Figure 48 Extend use case Relationship 66
Figure 49 Declaration of an object 68
Figure 50 Declaration of three objects instances of a class 69
Figure 51 Different objects with the same values of attributes 70
Figure 52 Operations in a class 71
Figure 53 Relationship between objects 71
Figure 54 Relationship between classes 72
Figure 55 No instance attributes in the class in the first time 73
Figure 56 Association between two classes 73
Figure 57 Multiplicities 74
Figure 58 Classes without inheritance 75
Figure 59 Classes with inheritance 75
Figure 60 Self-association 76
Figure 61 Example of an object diagram associated with the previous class diagram 76
Figure 62 Example of a multiple associations 77
Figure 63 Example of an associate objects diagram 77
Figure 64 Association class 78
Figure 65 Associate simple classes diagram 79
Figure 66 The associate objects diagram 79
Figure 67 The associate class diagram (without association classes) 80
Figure 68 Example of ternary association 81
Figure 69 Sequence Diagram 82
Figure 70 Synchronous message Source: BookMyEssay. Synchronous Message in Sequence Diagram. Accessed June 3, 2025 83
Figure 71 Asynchronous message Source: BookMyEssay. Asynchronous Message in Sequence Diagram. Accessed June 3, 2025 83
Figure 72 Role of sequence diagram 84
Figure 73 Creation and destruction of an object 85
Figure 74 Alternative bloc in a sequence diagram 86
Figure 75 Loop in a sequence diagram 86
Figure 76 Referencing the Diagram 87
Figure 77 Use case diagram 89
Figure 78 Use case and class diagrams 96

Chapter 1: Industrial Information Systems

1. Introduction

The term "Information System" (IS) is being used more and more frequently, whether through the acronym DSI (Information Systems Department) or, for example, in the reform of the final year of the IG program, which is now called GSI (Information Systems Management). But what exactly does an Information System represent, and what is its connection to computer science and industry?
In the context of organizations whether a business or a public institution like the state the concept of a "system" is fundamental. A system can be understood as a **set of dynamically interacting elements, organized with respect to a specific goal** (as defined by Joël de Rosnay in Le Macroscope, Éditions du Seuil). To achieve this goal, the system must both respond to and regulate itself according to its surrounding environment. This adaptation to change is made possible through the exchange of information flows. These flows not only connect the system to its environment but also circulate internally, enabling the system to monitor and optimize its own functioning. Within a company, all operational activities such as purchasing, production, sales, financing, and investment generate exchanges with external partners. These exchanges take the form of **flows**, which are essential for the system's coordination and performance.

2. The different flows

In any organization, three main types of flows are involved in operational processes: physical flows, monetary flows, and information flows.

2.1 Physical flows

Refer to the transfer of goods or services. This includes the purchase of raw materials, their movement through various production workshops, employee labor, the sale of finished products, or the purchase of services. Goods are tangible items such as raw materials, semi-finished, or finished products while services involve intangible work carried out by other entities, such as cleaning or consulting services. For example, when a furniture factory buys wood, nails, and paint, these raw materials enter the production system and become part of the manufacturing process. The labor of employees, whether on the shop floor or in administrative roles, is a crucial contribution to transforming inputs into deliverables, even if it does not constitute a direct physical flow.

2.2 Monetary flows

On the other hand, represent the movement of money within and beyond the organization. These include paying suppliers, receiving payments from clients, repaying loans to banks, and paying employee wages.

2.3 Information flows

Involve the circulation of information, either internally within the organization or externally with partners. For instance, a procurement manager might ask the production manager for the quantities of materials required for manufacturing (internal flow) in order to determine what to order from suppliers. The corresponding external exchanges such as purchase orders, invoices, or delivery notes represent structured information flows that support coordination and decision-making.

The figure below illustrates the interrelation between material, information, and monetary flows within a logistics system [1].

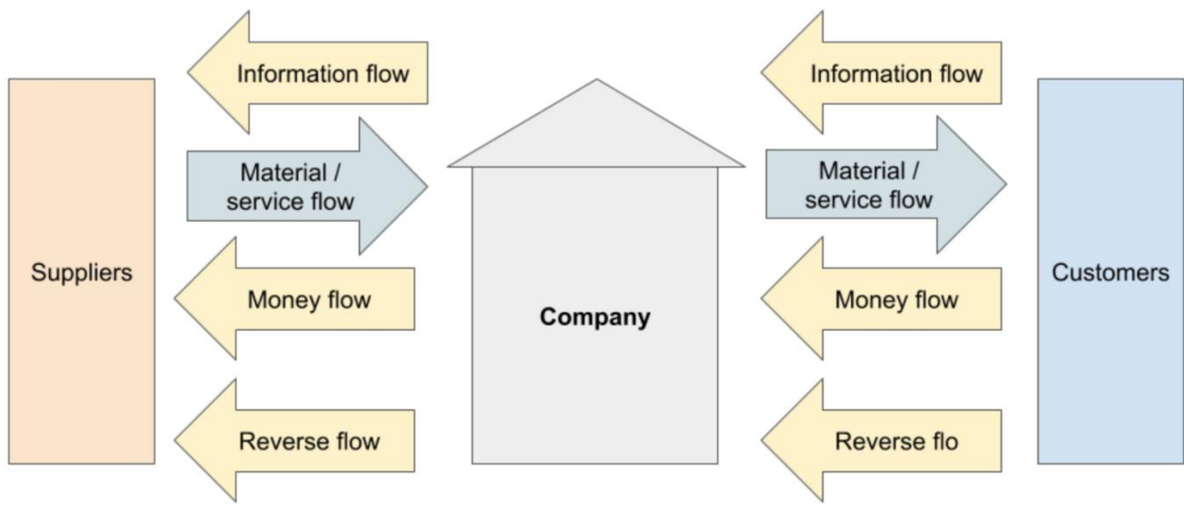


Figure 1 Information, money, and material flows in logistics.

Source: Logistiikan Maailma, https://www.logistiikanmaailma.fi/en/logistics/logistics-and-supply-chain/information-money-and-material-flow/, accessed on June 1, 2025.

2. The different subsystems within an organization:

An organization's Information System can be divided into two main subsystems: the **decision-making system** (also known as the control or management system) and the **operating system**.

The decision-making system is responsible for defining the strategic, tactical, and operational actions required to steer the organization toward its goals. It relies on data circulating through the information system to make informed decisions, such as launching a new product, entering a foreign market, or selecting the most suitable supplier. It also monitors the performance of the operating system and makes necessary adjustments such as increasing production capacity in response to market demand or reorganizing staff schedules during peak periods.

On the other hand, the **operating system** is in charge of executing the organization's core activities. This includes billing customers, paying employees, and managing inventories. It implements the directives issued by the decision-making system and, in doing so, generates operational data. This data then feeds back into the decision-making system, enabling it to evaluate performance and respond to deviations from planned objectives. The close interdependence between these two subsystems where information flows back and forth ensures that the organization remains responsive, efficient, and aligned with its strategic vision.

Within an organization, the **Information System** serves as a critical subsystem that supports the overall coordination and efficiency of operations. Its main role is to **collect, store, process, and disseminate information** essential to the organization's functioning. It gathers data from various sources internal operations, the external environment, or other internal systems and transforms raw data into meaningful insights to support decision-making. The system ensures that the right information reaches the right people at the right time. For instance, an ERP (Enterprise Resource Planning) system can track stock levels in real-time and alert managers to potential shortages, while a dashboard may display key performance indicators for executives.

The **Information System** is tightly interconnected with both the **decision-making system**, which it supports by providing reliable and timely data, and the **operating system**, which it informs and guides through instructions and operational data flows.

The **Operating System**, in contrast, is responsible for the **actual execution of activities** required to produce goods or deliver services. Its functions include transforming raw materials into finished products, delivering these products or services to customers, and carrying out all tasks defined by the decision-making system. For example, a production line manufacturing automotive parts is a core component of the operating system. This subsystem receives

instructions from the decision-making system regarding what needs to be done and provides feedback to the information system such as production volumes, machine breakdowns, or delays ensuring continuous monitoring and control of operations.

Together, these subsystems form a coherent and dynamic structure, where data, decisions, and execution are interlinked to maintain alignment with the organization's objectives.

3. Information system (IS)

To ensure effective operation, any organizational system must **store**, **process**, and **disseminate information** a function fulfilled by the **Information System (IS)**. The IS is a core component of any company or institution, enabling various stakeholders to **exchange information** and **communicate** through a combination of **human, technical, and software resources**. Its primary role is to **create, collect, store, process, and update information** in multiple formats to support decision-making and day-to-day operations.

An Information System is designed to **deliver the right information to the right person at the right time**, using the most appropriate format. It not only manages internal and external information flows within the organization but also provides the tools to compare, predict, and coordinate activities. A more operational definition sees the IS as the **sum of all information flows** within the organization, along with the **resources deployed to manage them**.

These resources include both **human and physical means** such as staff, computers, and automated systems. Although many IS functions are increasingly automated (with machines, software, and connected systems generating and processing data), **human input remains essential**, especially for tasks involving data entry, interpretation, or strategic decision-making. The IS thus plays a crucial role in ensuring that information is transformed into knowledge and action within the organization.

To process information, both computers and humans apply a set of **rules and procedures**. These may include **mathematical models, algorithms, standards, regulations, and administrative procedures**, all of which help structure how data is interpreted, transformed, and used within the organization.

3.1 The functions of an information system

Information Collection

For an organization to operate effectively, its Information System (IS) must be **fed with data** from various sources, both **internal and external**. External sources originate from the system's environment and typically include **partners such as clients, suppliers, and public institutions**. In today's fast-changing world, organizations must be highly responsive to their external environment in order to anticipate changes and adapt accordingly. The rise of communication technologies especially the internet has greatly facilitated access to information, though it also raises challenges related to **data quality and reliability**.

Internally, the IS is fueled by **information flows generated by the organization's own activities**, such as production, human resources, and accounting. While many of these flows are formalized through well-defined procedures, others are **informal**, such as social climate indicators or undocumented know-how. These informal flows, though harder to capture and analyze, can have significant strategic value.

Relevant information must be **recorded and entered into the system**, which is often a **costly process**, especially when it requires human input. This has led to growing efforts toward **automation**, including real-time systems and digitization technologies. Although information is a **critical asset** for organizations, its acquisition and management come at a cost making **efficiency in data collection and processing a major concern** for modern businesses.

Information Storage

Once information is collected, it must be **preserved reliably over time**. Ensuring the **durability and security of storage** is a core responsibility of the Information System. Today, the primary medium for storing information is digital using devices such as hard drives, solid-state drives (SSD), cloud storage, and other modern electronic storage technologies. Nevertheless, **paper-based storage** still holds a significant place in many organizations, especially for the **archiving of legal or administrative documents**.

Digitally stored information is usually kept in the form of **files** or organized into **databases** to facilitate access and processing. As such, the **Database Management System (DBMS)** becomes a **central component** of any modern Information System. In order to be processed and stored in a database, information must first be **converted into data**, since computers can

only handle data, not information per se. Conversely, the system must be capable of **reconstructing meaningful information** from this stored data when needed.

Information Processing

In order to be useful, information must undergo processing. While some processing tasks may still be done manually though increasingly rarely most are now automated and carried out by computer systems. The main types of processing include searching for and retrieving specific information, aggregating and consolidating data, comparing different sets of information, modifying or deleting existing data, and even generating new information through calculations or predefined rules. These operations are essential to transform raw data into meaningful insights that support decision-making and operational efficiency.

Information Dissemination

For information to be effectively used, it must **reach its intended recipient as quickly and efficiently as possible**. Various means are available to disseminate information: **paper documents, oral communication**, and most commonly today **digital media**, which ensure optimal transmission speed and broader reach. This is especially true in the era of **the Internet and interconnected information systems**, where rapid and widespread distribution of information has become both a necessity and a strategic advantage for organizations.

3.2 Qualities of an Information System

To be effective, an information system must ensure **fast and easy access to information**. **The speed and ease of access to information:** A system that is too slow or overly complex can discourage users, reduce operational efficiency, and negatively impact the quality of decisions.

Therefore, it is essential to rely on **high-performance hardware and networks**, as well as **user-friendly and practical interfaces** that facilitate interaction and adoption by all stakeholders.

Reliability, Relevance, and Integrity of Information: An effective information system must ensure that the data it provides is **trustworthy and up to date**. The **timeliness of data entry**, largely dependent on human actions, plays a crucial role in maintaining data reliability. On the technical side, the system must be **available whenever needed**, with maintenance operations ideally scheduled outside working hours.

Information integrity means the system must be able to detect and respond to issues that could lead to data inconsistency. For instance, if communication is interrupted between two computers that need to synchronize their data, the system must be capable of **restoring a** coherent state across both machines to preserve data accuracy.

Security and Confidentiality of Information: Data confidentiality is a key aspect of information system security. It ensures that only authorized individuals can access sensitive information.
Confidentiality can be maintained using hardware solutions such as card readers, fingerprint scanners, or facial recognition systems. It can also rely on software mechanisms, including user authentication procedures and permission settings for files or databases.

3.3 Examples of Information Systems

In large companies, the information system is often built around an **ERP (Enterprise Resource Planning)** system, which manages most of the company's operations. In French, this is referred to as a **PGI (Progiciel de Gestion Intégré)**. An ERP ensures centralized data management and consistency across various departments, facilitating efficient coordination and decision-making. However, certain specific functions of the company may be managed by other standard or custom-built information systems, depending on the organization's unique needs and operational characteristics.

In general, within production or service-oriented companies, different types of applications can be found, such as:

Commercial Management:

CRM (Customer Relationship Management): This includes all functions related to managing customer interactions and relationships. It is referred to in French as GRC (Gestion de la Relation Client).

MIS (Marketing Information System): A system used to collect and process data to support and guide the company's marketing strategies and decisions.

Human Resource Management (HRM):

This type of application allows companies to monitor employee careers, skills, training, salaries, leave, and other HR-related activities.

Logistics:

SCM (Supply Chain Management): This includes all functions aimed at integrating suppliers and logistics into the company's information system.

Geographic Management:

GIS (Geographic Information System): A GIS allows the production of maps, plans, and the geographic localization of sites, municipalities, and other spatial entities.

The most representative applications across different levels of the company include:

Control-command systems: These systems directly manage and control industrial machines and equipment in real time.

A **control-command system** is a system that **monitors and controls an automated process**. It works in three main steps:

1) **Measure:** It collects data from the environment (via sensors, cameras, etc.).
2) **Decide:** It analyzes the information and decides what action to take (using a program, AI, etc.)
3) **Act:** It sends commands to machines (motors, robotic arms, etc.) to perform the action.

Simple example: A thermostat

-It **measures** the room temperature.

-It **decides** whether to turn the heating on or off.

-It **sends a command** to the heater to perform the action.

In an ASRS (Automated Storage and Retrieval System), the control-command system manages product movements:

-It **detects the position** of items.

-It **selects the best product** to retrieve.

-It **sends instructions** to the machines to pick and move the item.

-It's like an **automatic brain** that makes decisions and gives orders to machines!

Supervisory systems: Used to monitor and visualize production processes, often through SCADA (Supervisory Control and Data Acquisition) platforms.
A supervision system is a system that **monitors and controls an automated process in real time**. Its main role is to **display information**, **detect anomalies**, and **support operators** in decision-making.

Difference with a control-command system:

Control-command system: Automatically **makes decisions and sends commands** to machines.

Supervision system: **Displays** what's happening and allows **human operators** to interact or intervene when needed.

How does it work?

1. **Data collection:** It receives information from sensors, PLCs, machines.
2. **Display and analysis:** It shows data on a screen (graphs, alarms, status indicators).
3. **Human intervention:** The operator can adjust parameters or correct issues.

Simple example: A car dashboard

- Displays speed, fuel level, engine temperature.
- Alerts you if there's a problem (red light, alarm).
- The driver reacts (slows down, refuels, etc.).

In an **ASRS** (Automated Storage and Retrieval System):

A **supervision system** could show:

-Rack status (full/empty)

-Products in motion

-Destocking/restocking times

-Errors or failures

In short: it’s the **interface between the automated system and the human operator**.

MES (Manufacturing Execution Systems): These systems bridge the gap between business planning (ERP) and factory floor operations. They monitor, track, and document the transformation of raw materials into finished goods in real time.
A **MES (Manufacturing Execution System)** is a key component of industrial information systems, whose **main goal** is to ensure that **production operations are executed efficiently** and to **enhance manufacturing performance**.

Main Functions of a MES:

Real-time production tracking → Monitors quantities produced, machine status, and production progress.

Efficiency improvement→ Helps reduce downtime, optimize material and information flows.

Product traceability→ Records who made what, when, how useful for quality control and audits.

Quality management→ Detects defects, verifies compliance with standards, and supports corrective actions.

Performance analysis→ Calculates KPIs like **OEE** (Overall Equipment Effectiveness) to identify improvement areas.

A MES connects the **shop floor (machines, operators)** with the **management level**, providing **real-time visibility**, **control**, and **data** to make better decisions.

CAPM systems (Computer-Aided Production Management): These support the planning, scheduling, and monitoring of production activities, including inventory management and work orders.Such a program allows to plan and monitor industrial activities, closely linked with planning, procurement, manufacturing, and sometimes distribution departments.
Main functions:

Production planning→ Anticipates needs for raw materials, machinery, and human resources.

Launching manufacturing orders→ Gives the green light to start production according to the defined schedule.

Batch size management→ Determines the optimal production lot sizes to minimize costs or meet deadlines.

Supplier order placement→ Automates purchases of raw materials or components needed for production.

Progress tracking→ Monitors production progress in real-time (delays, incidents, consumption, etc.)

PDM/PLM systems (Product Data Management / Product Lifecycle Management): Often referred to in French as SGDT (Système de Gestion des Données Techniques), these systems manage all data related to product design, engineering, and lifecycle.

Note: The four processes we just discussed are **process-oriented**, meaning they focus on the **manufacturing process** itself whether it is control, supervision, or command.In contrast, the **SGDT (Technical Data Management System)** is more **product-oriented**, focusing on managing the technical data related to the product rather than the manufacturing process.

3.4 Main Needs of Information Systems Integration (SII):

Integration mechanism: There is a need for different applications to interact with each other. This mechanism of interaction or cooperation between various applications is generally called an **integration mechanism**.

Typical examples of needs that may drive the use of integration within a company include:

-An application A that periodically and/or regularly needs to access data from application B;

-An application A that requires a function provided by application B;

-Applications A and B that need to share a common set of data to avoid redundancies;

-Applications A and B that must collaborate by exchanging data;

-Applications A and B that are orchestrated by an ad hoc application to carry out a business process.

Need for Flexibility: Organizations evolve, and these changes can significantly impact the information system. These changes may originate from:

-Internal sources (e.g., changes caused by organizational restructuring, creation of new subsidiaries, or new geographical locations).

-External sources (e.g., changes caused by mergers or acquisitions of organizations).

-Evolution of Regulations: These changes can impact the information system from an external (exogenous) source for example, the transition to the Euro currency or changes in calculation rules imposed by the government or local authorities. Such evolutions often require rethinking and/or modernizing the information system.

-Evolution of the Business: These changes are mainly due to the evolution of the company's activity (for example, the production of a new product line). Such business evolutions impact business processes and consequently affect the information system.

-Evolution of Technologies: These changes are mainly due to the evolution of software products, architecture models (mainframe, client-server, multi-tier, etc.), programming paradigms (C++, Java, etc.), and IT infrastructures (J2EE, .NET). These evolutions impact information systems as they often require efforts to integrate heterogeneous technologies.

-Cost Control: This is often a requirement imposed on IT departments, which are asked to reduce and/or control their costs. This most often translates into an effort to rationalize the information system.

4. Conclusion

This first chapter laid the essential foundations regarding information systems, defining their central role in the collection, processing, storage, and dissemination of data within organizations. An information system is much more than just an IT tool; it is an organized set of human, hardware, software, and procedural resources that contribute to decision-making, activity coordination, and value creation.

More specifically, industrial information systems are distinguished by their focus on automation, control, and optimization of production processes and resource management in industrial environments. They often integrate specialized technologies such as ERP (Enterprise Resource Planning), SCADA systems, and MES (Manufacturing Execution Systems), enabling fine synchronization between the production line, inventory management, and information flows.

In summary, mastering industrial information systems is a major strategic lever to enhance competitiveness, flexibility, and responsiveness of companies facing current technological and economic challenges.

For further study, one can refer to the classic work by Laudon and Laudon, which remains a key reference for understanding the impact of information systems in modern organizations [2].

Chapter 2: ERP (Enterprise Resource Planning) Systems

1. Introduction

In today's highly competitive and fast-paced industrial environment, companies must coordinate a wide range of business processes such as production, inventory management, sales, purchasing, accounting, and human resources in an efficient and integrated manner. To achieve this, many organizations rely on **Enterprise Resource Planning (ERP)** systems.

An ERP system is a comprehensive, modular software solution designed to centralize and streamline a company's data and operations across all departments. By offering a unified platform, ERP systems enhance decision-making, improve productivity, and ensure consistency throughout the organization.

This chapter provides an overview of ERP systems, starting with their **definition**, **key functionalities**, and **main characteristics**. We will then explore the **different types of ERP systems**, from proprietary to open-source solutions. Special attention will be given to **ODOO**, a widely used open-source ERP platform, which we will examine in more detail to understand its architecture, modules, and practical applications in industrial settings.

By the end of this chapter, students will gain a solid understanding of how ERP systems function and why they are critical tools for modern industrial enterprises.

2. What is an ERP?

ERP stands for **Enterprise Resource Planning**, which is the English equivalent of the French term **PGI (Progiciel de Gestion Intégré)**.

An ERP is an integrated software solution designed to manage **all of a company's processes** by incorporating all its functions (Figure 2) such as **human resources management**, **accounting and financial control**, **decision support**, as well as **sales**, **distribution**, **procurement**, and **e-commerce**.

Figure 2 Different Modules in an ERP System

The main objectives of an ERP system are:

-To **network** all company functions through a **single, unified database**;

-To **centralize and harmonize** information flows across departments;

-To **increase the overall performance** and efficiency of the enterprise;

By providing real-time access to reliable data, ERP systems support better decision-making, improve coordination, and reduce redundancies.

3. Principle of ERP Systems

ERP systems are a **set of configurable software applications**, organized into **independent modules** that each manage a specific business function.

While these modules operate independently, they **share a single, centralized database [3]**, ensuring data consistency and real-time information flow across the entire organization (**Figure 3**).

This modular and integrated architecture allows companies to adapt the ERP system to their specific needs while maintaining **coherence and unity of information**.

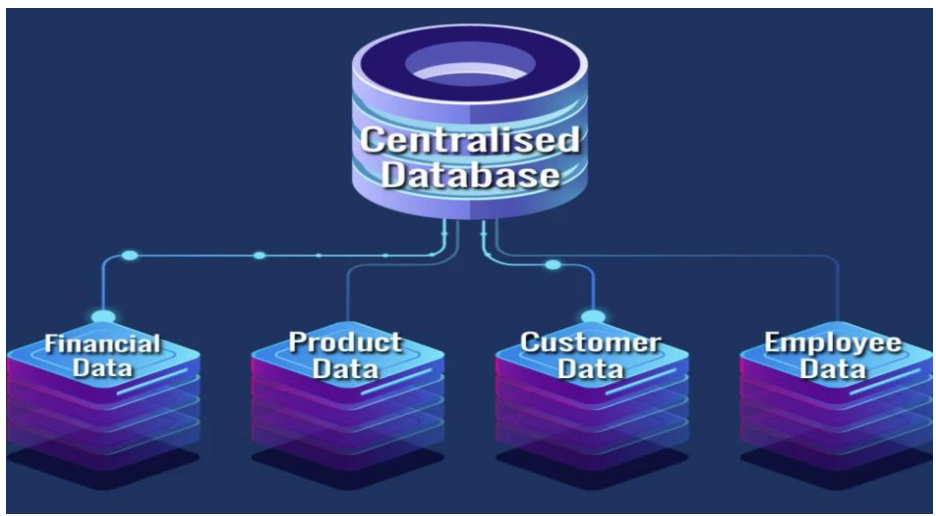


Figure 3 Centralised database architecture
***Source*: SOLIDitech. (2021, July 13). *An Introduction to Centralised Databases* [Image]. The SOLID Blog. https://blog.soliditech.com/blog/an-introduction-to-a-centralised-databases**

4. ERP Functionalities

ERP systems offer a wide range of functionalities that cover the core activities of an enterprise. These include, but are not limited to:

- **Marketing Management**
- **Human Resources Management**
- **Supply Chain Management**
- **Sales Management**
- **Customer Relationship Management (CRM)**
- **Financial and Accounting Management**
- **... and many other business processes**

Thanks to its modular structure, an ERP system allows each company to activate only the functionalities that correspond to its specific operational needs, while still benefiting from an integrated and unified information system.

5. Key Characteristics of an ERP System

ERP systems are defined by several essential characteristics that make them powerful tools for managing complex organizations:

-Use of a Single, Centralized Database→All modules share a common database, ensuring data consistency and real-time information access.

-Integrated Functionalities Across All Management Areas→From finance and HR to sales, logistics, and customer service, an ERP covers the full range of business functions.

High Configurability→ERP systems offer a large capacity for customization and parameterization to fit the specific needs of each organization.

Open and Scalable IT Architecture→Modern ERP systems are designed with open architecture, allowing for interoperability, scalability, and integration with other software tools.

Speed and Efficiency→By streamlining processes and eliminating redundant tasks, ERP systems improve productivity and response times.

Cost and Time Savings→Centralized data and process automation reduce operational costs and speed up workflows because time is money.

Decision Support Tool→ERPs provide analytical tools, dashboards, and reporting functions to assist managers in making informed decisions.

Traceability and Data Tracking→ERP systems ensure full traceability of operations and transactions, enhancing control, accountability, and quality assurance.

5.1 Use of a Single Database: Historical Evolution of Management Information Systems

The evolution of management information systems has gone through several key stages:

Independent Applications: In the early stages, each business function (such as accounting, HR, inventory, etc.) relied on its own software application **Figure 4**, often with a separate database. This led to redundant data entry, inconsistent information, and limited communication between departments.

Figure 4 Independent Applications

Integration into an ERP System: With the advent of ERP systems, these separate applications were replaced by a **modular system** based on a **shared, centralized database Figure5**.

This integration ensures data consistency, eliminates duplication, and facilitates real-time access to accurate information across all departments.

This transition marked a major step toward the **digital unification of business processes**, significantly improving coordination and decision-making within the organization.

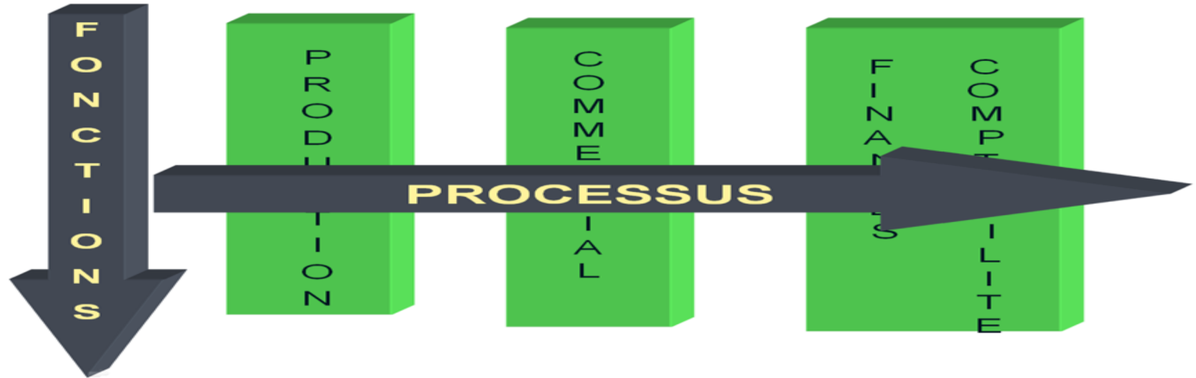


Figure 5 Integration into an ERP

5.2 Integration into an ERP System

In an ERP system, **each piece of data is entered only once**, at the moment the event that generates it occurs.

This data is then made **immediately available in real time** to all authorized users across the organization, through a **shared, common database** used by all modules.

The use of a **relational database** eliminates redundancies, ensures data consistency, and facilitates seamless information flow between departments.

5.3 ERP Modules Cover All Business Activities

ERP modules are designed to **cover the entire range of a company's activities**. The **richness of their functionalities**, combined with **integrated data**, allows for a **comprehensive and unified approach to management**.This integration facilitates better coordination between departments, improved decision-making, and an overall increase in organizational efficiency.

5.4 High Configurability

ERP systems offer a **high level of configurability**, allowing companies to tailor the system to their specific industry and operational practices.

-Database configuration can be adapted to the company's business logic and workflows.

-User interface customization is available for each workstation, including:

-Access rights and permissions

-Personalized graphical environment

-Operation traceability and audit trails

Example:

The textile company Gilclaude needed to manage a product table containing 300 to 400 items, each with about twenty variants (size/color).

Thanks to advanced configuration, the company avoided the multiplication of item references by implementing a lot-based management system using generic product entries.

This kind of setup significantly simplifies inventory tracking and data maintenance, while keeping the system aligned with the company's real-life practices.

5.5 Decision Support Tool

An ERP system provides **management control** with powerful tools to support strategic and operational decision-making. It enables:

-The use of **SQL query languages** from relational databases such as Oracle, SQL Server, or DB2

-Client/server access to the shared database, allowing multi-user and remote interactions

-The **creation of customized reports and dashboards**, tailored to users' specific needs

These capabilities make the ERP a valuable tool for analyzing performance indicators, monitoring operations, and supporting informed, data-driven decisions across the organization.

6. Types of ERP Systems

ERP systems can generally be classified into two main categories:

Proprietary ERP Systems: These are commercial solutions developed and distributed by software vendors. Their source code is closed, and they typically require the purchase of licenses and ongoing maintenance contracts (e.g., SAP, Oracle ERP, Microsoft Dynamics).

Open Source ERP Systems: These are ERP solutions whose source code is publicly available and modifiable. They offer greater flexibility and can be customized extensively, often at a lower cost (e.g., Odoo, ERPNext, Dolibarr).

Cloud-Based ERP Systems (Cloud ERP): These ERP systems are hosted on remote servers and accessed through the internet. Instead of installing the software locally, organizations subscribe to the service, typically on a pay-as-you-go basis. Cloud ERP solutions offer advantages such as scalability, lower upfront costs, automatic updates, and remote accessibility. However, they may raise concerns regarding data security, internet dependency, and limited control over system customization.

Each type has its own advantages and drawbacks depending on the size, budget, and technical capacity of the organization (**Figure 6).**

Figure 6 Distribution of leading ERP vendors in the market

7. The Odoo ERP System

Odoo, formerly known as **OpenERP** and **Tiny ERP**, is an open-source integrated management software that includes a large number of modules designed to simplify the overall management of a company.

With **over two million users worldwide**, Odoo is currently the **most popular open-source ERP system**.

Odoo offers two versions:

-Odoo Community Edition (Free) →This version is open-source and includes a wide range of essential applications for managing a business.

-**Odoo Enterprise Edition (Paid)** →This version includes all features of the Community edition, plus additional advanced applications and services such as:

Bank interfaces, Electronic signature, Integration with third-party platforms like **eBay** and **Amazon**

Thanks to its modular and user-friendly design, Odoo is widely adopted by companies of all sizes, from startups to large enterprises.

The concepts presented in this course are based on Odoo 9 for pedagogical reasons. However, they remain fully applicable to more recent versions of Odoo (Odoo 14, 15, 16, and beyond), with some minor adaptations related to changes in the interface and certain technical commands.

7.1 Odoo Architecture

Odoo is based on the **MVC architecture** (**Model-View-Controller**), a common design pattern in software development that separates concerns into three distinct layers:

Model

This layer manages the **data structure** and is linked to the **database**.Each object declared in Odoo corresponds to a **model**, which is mapped to a **table in PostgreSQL**.

View

This layer defines the **user interface**. Odoo uses **XML files** to design the views (forms, lists, dashboards, etc.) that users interact with.

Controller

This layer is handled by **Python classes** using Odoo's **ORM (Object-Relational Mapping)** system.Controllers manage the business logic and communication between the model and the view.

This architecture allows Odoo to be **modular, scalable, and easily customizable**, making it adaptable to a wide variety of business needs.

Figure 7 illustrates the MVC architecture of Odoo.

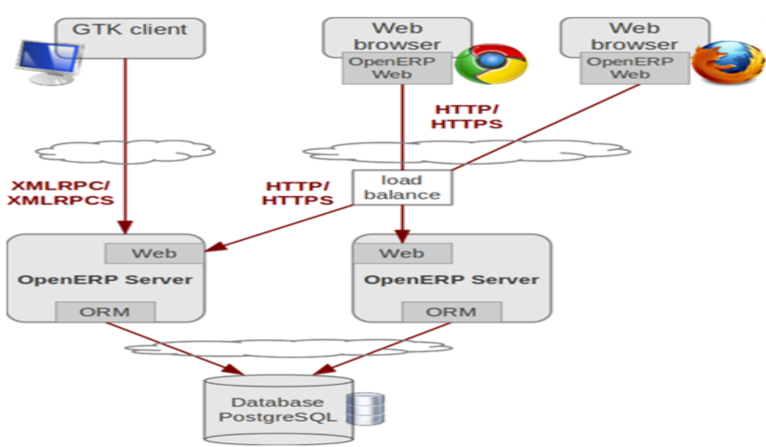


Figure 7 The MVC architecture

Modular Architecture of Odoo: Odoo's architecture allows for the development of software applications in a **modular way**, meaning that each module operates **independently** from the others while sharing a **common, single database (Figure 8)**.

This approach **eliminates multiple data entries** and **removes ambiguity** caused by duplicate or inconsistent data of the same kind, ensuring data consistency and integrity across the system.

Figure 8 Modular Architecture of Odoo

7.2 ORM: Object-Relational Mapping

Object-Relational Mapping (ORM) is a technique that establishes a link between a **class** (in object-oriented programming) and a **table** (in a relational database), as well as between each **attribute of the class** and a **field in the associated table**.

For example, the class Customer would be mapped to the table CUSTOMER, with the attributes associated as follows:

Customer ≈ CUSTOMER

Customer.customerId → CUSTOMER.CUSTOMER_ID

Customer.customerName → CUSTOMER.CUSTOMER_NAME

Customer.customerAddress → CUSTOMER.CUSTOMER_ADDRESS

This mapping allows developers to manipulate database records as Python objects, simplifying data access and logic implementation in systems like **Odoo**, which uses an ORM to manage models.

7.3 Languages and Technologies Used in Odoo

Python: is an interpreted, object-oriented programming language.(It is **interpreted**, meaning there is no explicit compilation phase.)

It was developed by **Guido van Rossum** at the **CWI (Centrum Wiskunde & Informatica)** in the Netherlands.

One of Python's key strengths is its **cross-platform compatibility** it runs on most common operating systems (Unix, Windows, etc.).

Python is also an **open-source language**, supported, developed, and widely adopted by a large global community of developers.

It is **widely used in web development** due to its simplicity and flexibility. Python's **clean and readable syntax** is very close to human language, which makes it **easy to learn and use**, even for beginners.

XML (eXtensible Markup Language): stands for **eXtensible Markup Language**. It is a computer language used to **store and structure textual data**.

XML is **self-descriptive**: the tags are **not predefined**, allowing developers to define their own tags based on the structure of the data.

It is a **text-based data format**, which means any XML file can be opened and read using a **simple text editor**.

An XML file consists of **tags** that are made up of **pure text**, organized in a hierarchical structure.

In Odoo, XML is widely used to define:**User interface views** (forms, lists, kanban, etc.), **menus and actions** and **access rights and workflows.**

HTML vs XML

HTML (HyperText Markup Language) →HTML is a markup language that uses **tags** to structure documents viewable on the **web**.

HTML is mainly used to structure content, while presentation is handled separately using CSS, which allows better management and reuse of data.

XML (eXtensible Markup Language) →Unlike HTML, XML provides a **strict separation between content and presentation**. It focuses on **storing and structuring data**, making it ideal for data exchange between systems.

XML Syntax

An XML document is considered **well-formed** if it follows these rules:

1) XML documents must begin with an **XML declaration**.
2) XML documents must have a **single root element**.
3) All XML elements must have a **closing tag**.
4) **Tags are case-sensitive**.
5) XML elements must be **properly nested** (no overlapping tags).
6) XML attribute values must always be **enclosed in quotation marks** (either single ' or double " quotes).
7) **Entities must be used** for special characters (e.g., & for &, < for <, etc.).

<u>Note</u>: One should not confuse the notion of a **well-formed XML document** with that of a **valid XML document**. A **valid** XML document (**Figure 9)** is a well-formed document whose structure **complies with a predefined model**, described by a **DTD (Document Type Definition)** or an **XML Schema**.

DTD (Document Type Definition): defines the **grammar** or **structure rules** of the XML document.

```xml
<?xml version = "1.0" encoding = "ISO-8859-1" ?>

<!DOCTYPE bibliothèque SYSTEM "bibliothèque.dtd">

<bibliothèque>
  <livre catégorie = "BD">
    <titre>Gaston Lagafe</titre>
    <auteur>Franquin</auteur>
    <pages>61</pages>
  </livre>
  <livre catégorie = "Roman">
    ...
  </livre>
</bibliothèque>
```

Figure 9 Example of a valid xml file

DTD, XML Schema, XSL, and CSS:

A **DTD** (Document Type Definition) is either a **file** or a **part of an SGML or XML document** that describes the structure of the document. A DTD defines the **grammar** of the document listing the **elements (tags)**, **attributes**, their **content types**, and their **arrangement**.It can also include **a vocabulary of character entities**. ***Example***: a tag <age> may be required to contain a **positive number less than 105**.

An **XML Schema** is a **more sophisticated alternative** to DTDs.It provides a richer set of data types and constraints, and it is written in XML syntax, unlike DTDs.

XSL (eXtensible Stylesheet Language) refers to **stylesheet files** used to **transform** XML documents into other formats and to **format the output**.It enables the transformation of XML into HTML, PDF, or other structures.

CSS (Cascading Style Sheets) was created in 1996 to apply **style and formatting** to content. CSS can be used with XML to **visually format** data (e.g., using **colors**, fonts, spacing, etc.). Unlike XSL, which transforms content, CSS **styles the existing content** for display.

PostgreSQL

Is an **object-relational database management system (ORDBMS)**.It supports the use of **object-relational mapping (ORM)** to link classes and attributes in the application code to tables and fields in the database.

pgAdmin: a **graphical administration tool** for PostgreSQL, distributed under the **PostgreSQL license**.It allows users to **manage databases**, **execute SQL queries**, and **visualize structures** via a user-friendly interface.

7.4 Structure of an Odoo Module

All modules are located in the server/addons directory.

To create a new module, the following steps are required:

Create a subdirectory inside the server/addons folder.

Create a module description file: __manifest__.py (formerly __openerp__.py) → This manifest file contains the metadata and configuration of the module.

Create an __init__.py file to load the Python files and import objects.

Create the main Python file containing the model classes (business objects).

Create XML view files to define the user interface (forms, trees, etc.).

Additional directories and files may include:

wizard: Contains transient models used in wizard interfaces→ These are temporary interactive windows that help the user perform operations such as calculations or data entry. The data is deleted automatically after use.

workflows: Defines business processes and their sequential steps.

report: Contains **QWeb or XML templates** for generating PDF reports.

security: Defines **access rights, user groups, and security rules** through .csv or .xml files.

static: Stores **images, logos, stylesheets**, and static web assets.

i18n: Contains **translation files** (.po) to internationalize the module

7.5 Practical Application and Exercises

To complement the theoretical concepts presented in this chapter, a series of guided exercises is proposed. These exercises are designed to progressively introduce learners to the practical use of Odoo, focusing on its core components and development mechanisms.

The sequence begins with an exercise dedicated to the **creation of a basic Odoo module**, allowing learners to understand its structure, configuration files, and modular architecture. This step provides a foundation for working within the Odoo development environment.

The next set of exercises focuses on the **model layer**, where learners define data structures by creating business objects and exploring different types of fields, including relational fields. This stage highlights the role of the ORM in managing data within Odoo.

Subsequently, learners explore the **presentation layer**, where they design user interfaces using XML by defining views, menus, and actions. These exercises demonstrate how data is displayed and interacted with in the system, reinforcing the link between backend structures and frontend representation.

Finally, the last exercises introduce the concept of **inheritance in Odoo**, enabling learners to extend existing models and views. This allows them to reuse and customize existing functionalities, which is a key aspect of Odoo development.

Through this progressive approach, learners gain hands-on experience with the essential components of Odoo, from module creation to advanced customization techniques, thereby strengthening their understanding of ERP systems in a practical context.

7.5.1 Exercise 1: Creating an Odoo Module

Objectives

-Install Odoo 9 Community Edition (free version)

-Understand the MVC (Model–View–Controller) architecture

-Understand the modular architecture

-Understand the structure and components of an Odoo module

-Use the *scaffold* command

Odoo, formerly known as OpenERP and TinyERP, is an open-source Enterprise Resource Planning (ERP) system that integrates a large number of modules designed to simplify overall business management. It is used by more than five million users worldwide and is currently the most popular open-source ERP system.

Two versions of Odoo are available:

- **Odoo Community Edition (free):** the version used in this exercise.

- **Odoo Enterprise Edition (paid):** provides additional applications and advanced features not available in the Community version, such as bank interfaces, electronic signatures, and integrations with third-party platforms (e.g., eBay, Amazon).

In this exercise, we will use **Odoo version 9**, which is sufficiently stable and feature-rich for learning and practical implementation.

Tasks

1. Create a new database from the main Odoo interface.
2. Create your first module using the *scaffold* command.
3. Analyze in detail the structure and components of this module.
4. Access and explore your database using the *pgAdmin* tool.

Odoo Architecture

Odoo follows a **three-tier architecture**, composed of the following layers:

-Database Layer: responsible for data storage.

-**Application Layer:** responsible for data processing and business logic.

-**Presentation Layer:** provides the user interface.

These layers are distinct and work together to ensure the proper functioning of the system.

Modular Architecture of Odoo

Odoo is based on a **modular architecture**, which allows the development of software applications as a set of independent modules (Figure 1), all sharing a single database.

This approach eliminates redundant data entry and reduces inconsistencies or ambiguities related to duplicate data, ensuring better data integrity and consistency across the system.

Once Odoo is installed, the database creation page will be displayed (Figure 10). Enter the required information in order to create a new database.

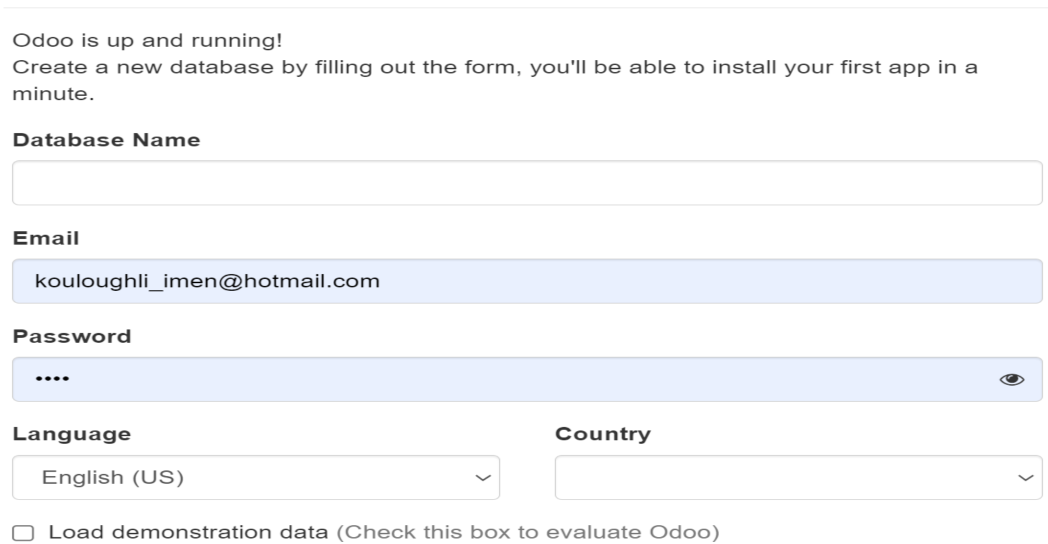


Figure 10 Creating a database with Odoo

Structure of an Odoo Module

An Odoo module is composed of at least two files and two directories (Figure 11).

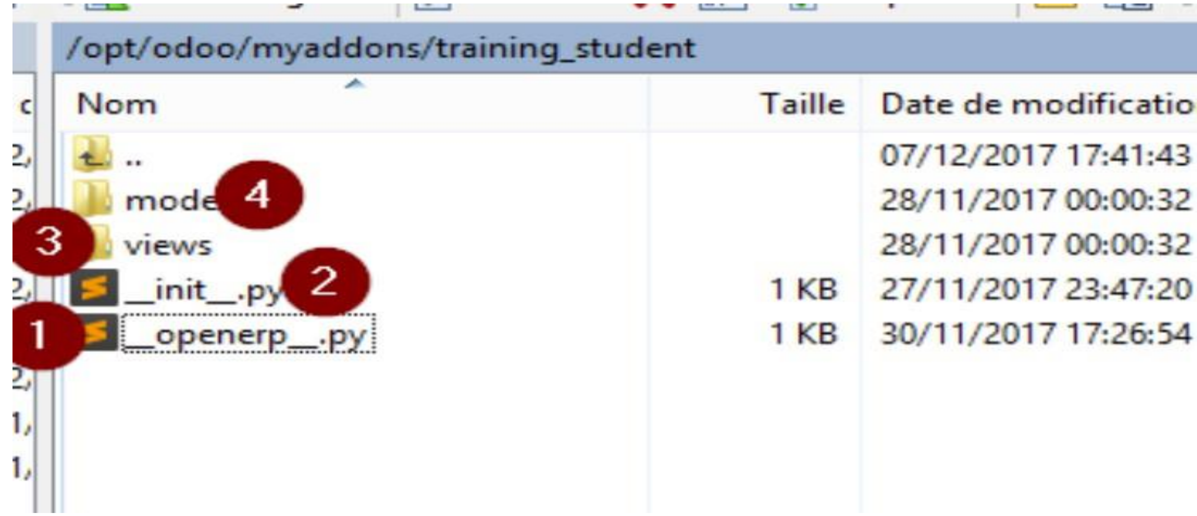


Figure 11 Structure of an Odoo module

1. **Manifest file:** describes the module
2. **Init file:** used to load Python files
3. **XML views:** define the user interface
4. **Python files:** contain the business logic

The Manifest File

The Manifest file is used to declare a Python package as an Odoo module and to specify the module’s metadata (Figure 12).

This file is generally named `__openerp__.py`.

It allows developers to configure and customize the module declaration.

```
{
'name':"A Module",
'version':'1.0',
'depends':['base'],
'author':"Author Name",
'category':'Category',
'description':"""
    Description text
    """,
# data files always loaded at installation
'data':[
'mymodule_view.xml',
],
# data files containing optionally loaded demonstration data
'demo':[
'demo_data.xml',
],
}
```

Figure 12 Example of a Manifest file

Main Fields of the Manifest File

The main fields of the Manifest file are as follows:

name (str, required): The name of the module.

version (str): The version of the module.

description (str): A structured description of the module.

author (str): The author of the module.

website (str): The author's website.

license (str, default: LGPL-3): The license of the module.

category (str, default: Uncategorized): The classification category of the module in Odoo, representing its functional domain. Although using existing categories is recommended, this field is flexible, and new categories can be created dynamically. Category hierarchies can be defined using the "/" separator. For example, *Education/University* creates a parent category *Education* and a subcategory *University*, with *University* assigned as the module's category.

depends (list(str)): The list of Odoo modules that must be loaded before this module, either because it uses or extends their functionalities. When a module is installed, all its dependencies are installed beforehand. Similarly, dependencies are loaded before the module itself.

data (list(str)): A list of data files that must always be installed or updated with the module. These are specified as paths relative to the module's root directory.

Example: 'data': ['views/student_view.xml'],

installable (bool, default: True): If this parameter is not set to True, the module will not display an install button in the applications list.

auto_install (bool, default: False): If set to True, the module will be automatically installed when all its dependencies are installed.

This option is generally used for *link modules*, which implement integration between two otherwise independent modules. For example, the *sale_crm* module depends on both *sale* and *crm* and is configured with auto_install = True. When both modules are installed, CRM campaign tracking is automatically added to sales orders, even though there is no direct dependency between the *sale* and *crm* modules.

The Init File

The **Init file** must contain import statements. This means that in the __init__.py file, all Python files used in the module must be imported.

For example, if the Python files are located in the *models* directory (as shown in Figure 13), the following line should be added in the __init__.py file:

```
__openerp__.py ×   __init__.py ×
1  # -*- coding: utf-8 -*-
2
3  import models #pour importer tous les objets qui se trouvent dans le dossier models de notre module
4
5
```

Figure 13 Init File

Views

This directory contains the XML files used to structure data for display in the browser (Figure 14).

The Models Directory

This directory contains the Python files used to define classes, functions, and fields.

It includes all the classes that will later be mapped to tables in a relational database (Figure 14).

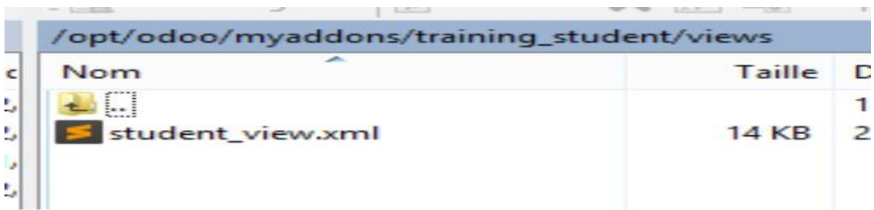


Figure 14 Views File

Using the Scaffold *Command*

You will now create a new module in the addons directory (which contains all modules developed by the Odoo community).

Using the command line (CMD), make sure you are running in Administrator mode. First, navigate to the server directory, as this is where the Odoo server is located. To do this, use the cd (change directory) command followed by the path to the server folder.

Your objective is to start the Odoo server (openerp-server.exe).

Once the server is running, use the *scaffold* command, as shown in the following figure (Figure 15), to create a new module named **testmodulenum1**.

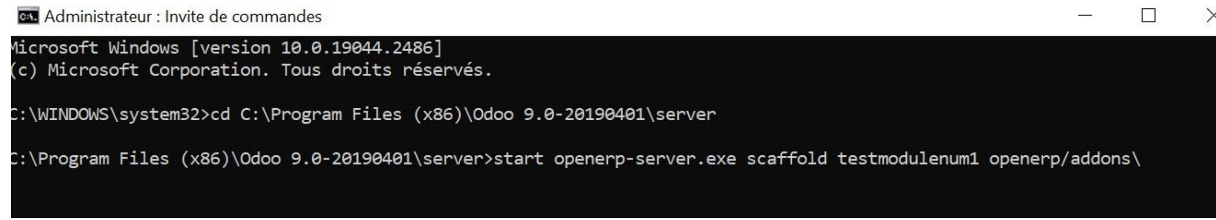


Figure 15 Using the scaffold command scaffold command

Verify that your module has been successfully created and added to the *addons* directory (Figure 17).

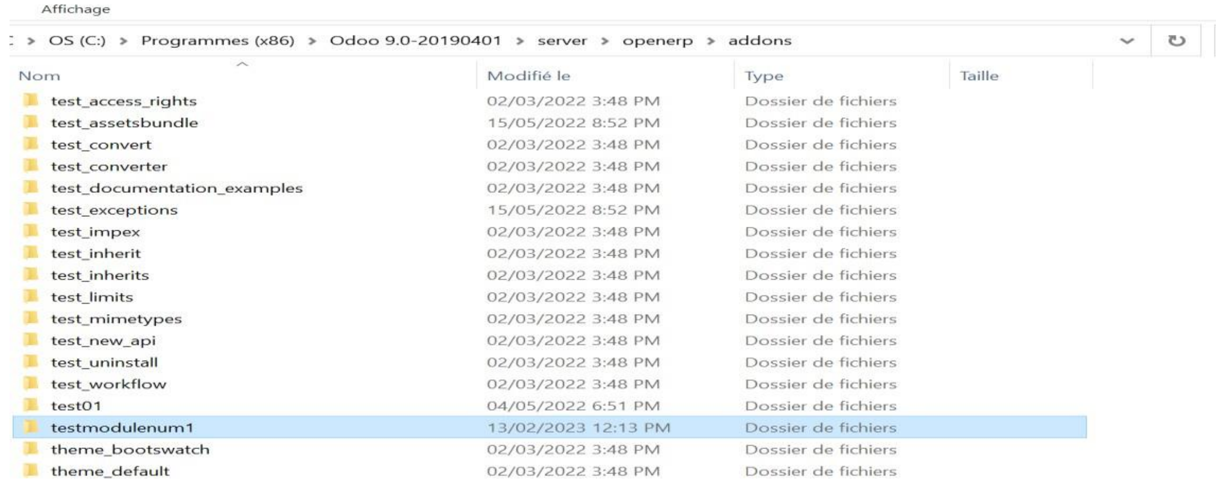


Figure 16 Module added to the addons directory

You should now verify that your module has also been successfully added to the main Odoo interface.

To launch the Odoo main interface, you need to use a web browser; however, it is recommended to use **Google Chrome**.

Since the application server is installed on your local machine, you will connect locally using: **localhost**.

The default port number is **8069**. You can see what the Odoo main interface looks like in the figure below (Figure 17).

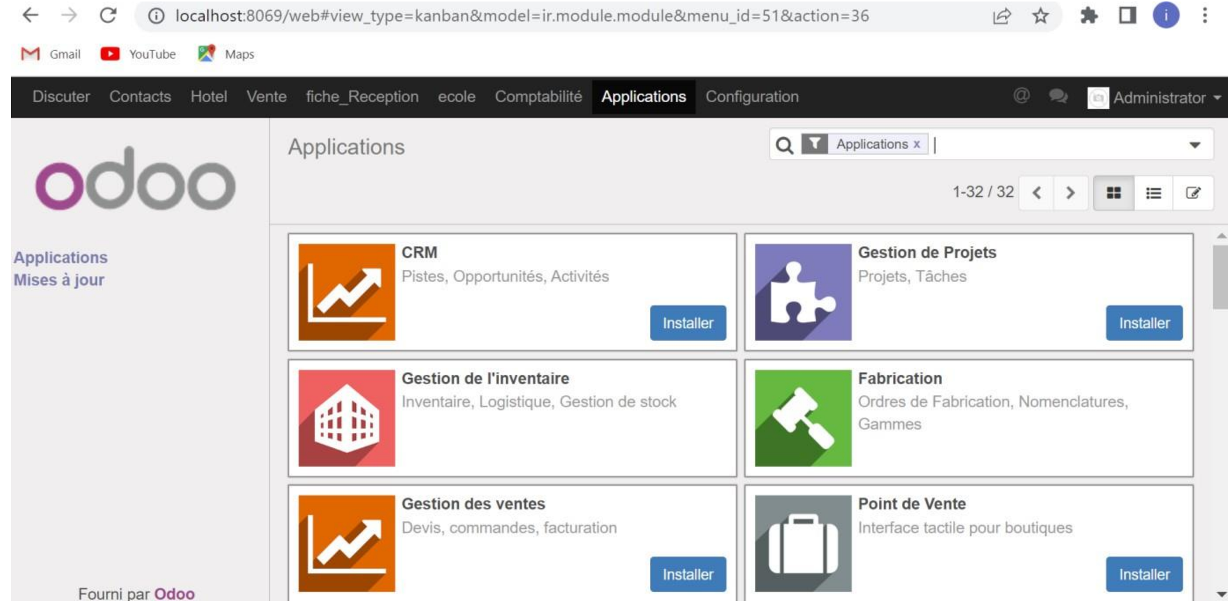


Figure 17 Odoo Main Interface

If you search for the module you have just created (*testmodulenum1*) among the other modules, you will not find it. This is normal.

You must first restart the Odoo server so that it takes into account the changes you have made (Figure 18).

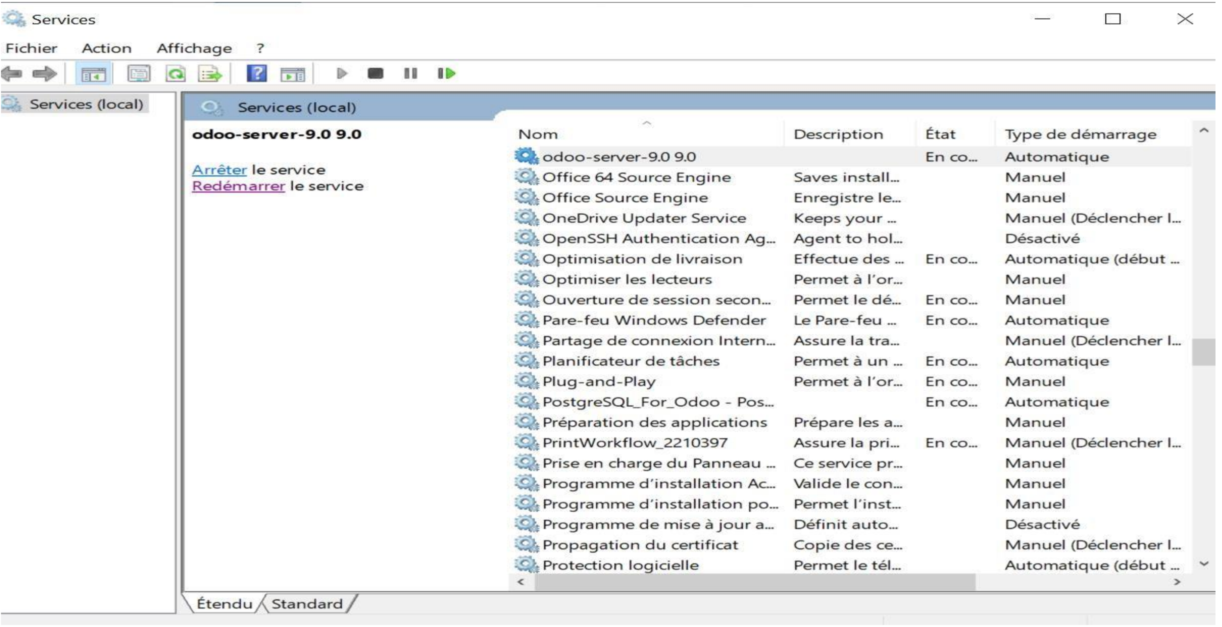


Figure 18 Restarting the Odoo server

You must now activate the **Developer Mode** and update the list of applications, as shown in the following figures (Figure 19, Figure 20, and Figure 21).

Figure 19 Accessing Developer Mode

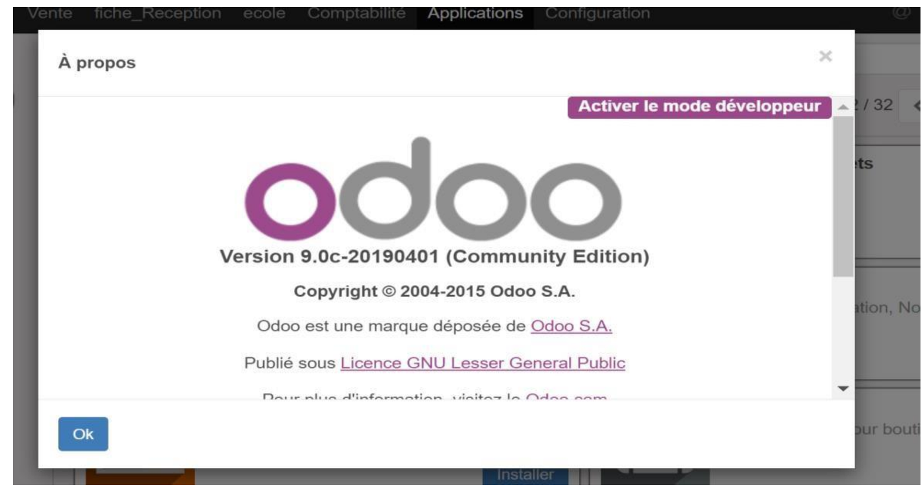


Figure 21 Activating Developer Mode

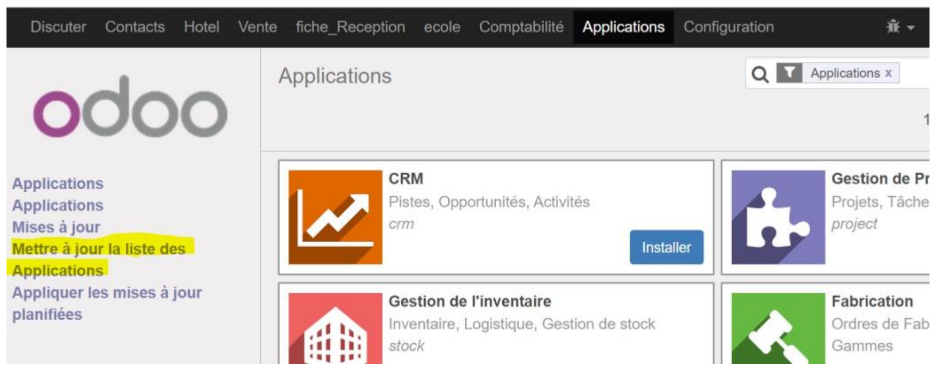


Figure 20 Updating the applications list

Search for your module again, and you will find it listed among the other modules (Figure 22).

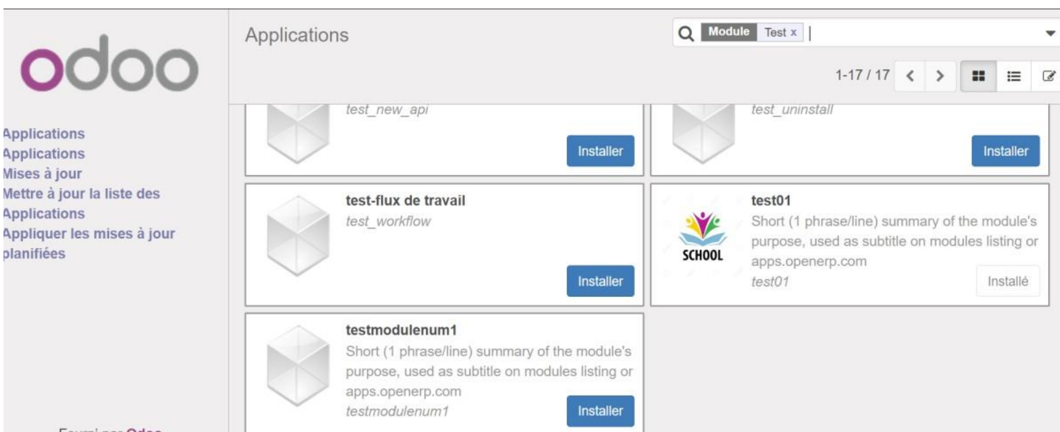


Figure 22 Display of the created module in the Odoo main interface

Note:

You can access and explore your database using the *pgAdmin* tool.

You will need to enter the password **openpgpwd**, provided during the Odoo installation (Figure 23 and Figure 24).

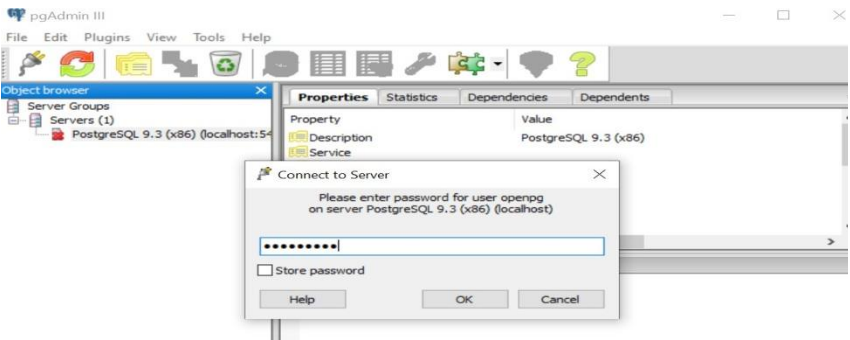


Figure 24 Entering the password

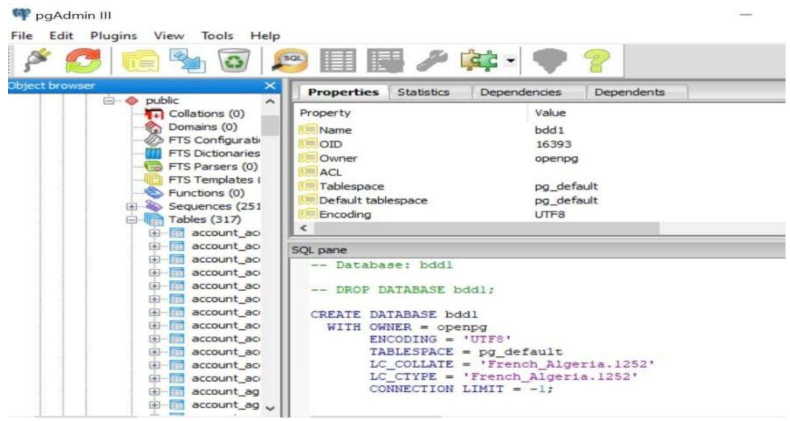


Figure 23 Relational database

7.5.2 The Model Layer in Odoo

Objective

Developing in Odoo most often involves creating custom modules. Odoo follows an **MVC (Model–View–Controller)** architecture:

Model layer: defines the data structure of the application

View layer: describes the user interface

Controller layer: handles the business logic of the application

The objective of this exercise is to learn the fundamentals of the **model layer**.

Now that Odoo recognizes our new module (Exercise 1), we will begin by adding a simple model.

Models represent business objects such as opportunities, sales orders, or partners (customers, suppliers, etc.). A model consists of a set of attributes and may also define specific business logic.

Models are implemented using Python classes that inherit from Odoo's model classes. They are automatically mapped to database objects, meaning that Odoo handles their creation and management in the database during module installation or upgrade. This mechanism relies on **Object-Relational Mapping (ORM)**.

Note:
Odoo requires a database to store all information related to your instance. You can create multiple databases for different companies or for different purposes (e.g., testing, production).

Step 1: Creating a Data Model

Create a file named student.py in the *models* directory using the code shown below (Figure 25).

```python
#!/usr/bin/env python

# -*- coding: utf-8 -*-

from openerp import models, fields,api

class Etudiant(models.Model):
    _name = "etudiant"

    Nom=fields.Char("Nom:")
    age=fields.Integer("Age:",help="Age")
    WilayaNaissance=fields.Char("Wilaya de Naissance")
    Filiere=fields.Char("Filiere")
```

Figure 25 Model creation file

1. # -*- coding: utf-8 -*- → This is a special marker indicating to the Python interpreter that the file is encoded in UTF-8, allowing it to handle non-ASCII characters.
2. The second line is a Python import statement that makes the **models** and **fields** objects from the Odoo framework available.
3. The third line declares our new model. It is a class that inherits from models.Model.
4. The following line defines the attribute _name = "etudiant", which serves as the identifier used by Odoo to reference this model. Note that the actual Python class name (*Etudiant*, in this case) has no significance for other Odoo modules. The _name value is what is used as the unique identifier.
5. The last lines define the fields of the model (e.g., *char* and *integer* types).

Note

The following declaration: name = fields.Char("Description", required=True) means that, by default, Odoo will use the **name** field as the display label of the record when it is referenced from other models.

At this stage, the model file student.py is not yet used by our module. We must instruct Python to load it by modifying the __init__.py file.

Add the following line (Figure 26):

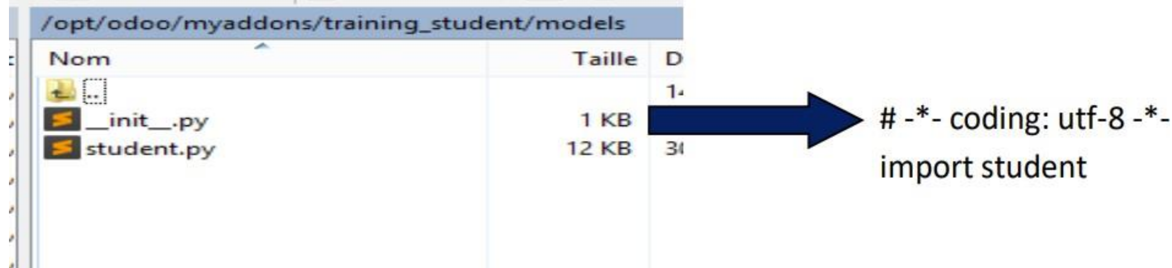


Figure 26 Loading the student.py file

Step 2: Installing the New Module

The **Developer Mode** must be activated in order to access the technical settings of Odoo (you can refer to the previous exercise to see how to activate it).

From the main **Applications** menu, select the **Update** option. This will refresh the list of modules by adding all modules that have been created since the last update. Enter the name of your module in the search bar, and you should see your new module ready to be installed.

After installation, the newly created model can be inspected using the **Technical** menu. From the top menu, go to:

Configuration → **Technical** → **Database Structure** → **Models**, then search for the model **"etudiant"** in the list and click on it to view its definition (Figure 27).

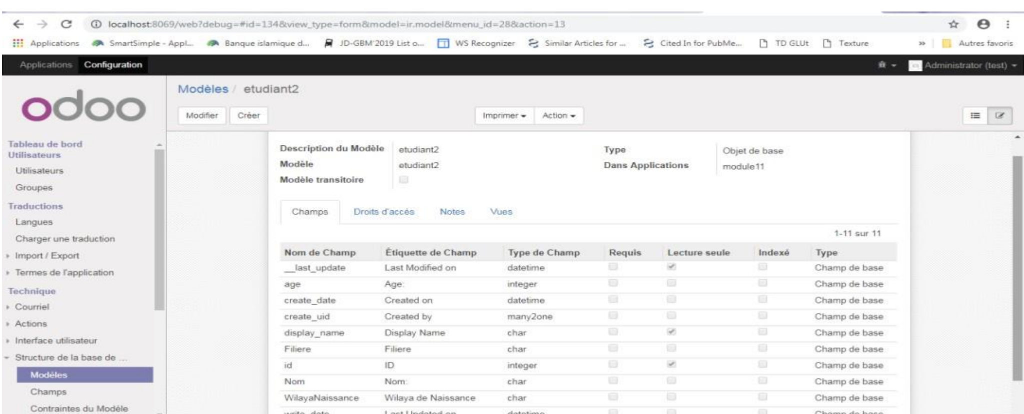


Figure 27 Viewing the created model

If everything is correct, it confirms that the model and its fields have been successfully created. If you cannot see them, try restarting the server and updating the module.

We can also observe additional fields that were not explicitly defined. These are reserved fields automatically added by Odoo to every new model. They are as follows:

id: a unique numerical identifier for each record in the model

create_date and **create_uid:** indicate the date and the user who created the record

write_date and **write_uid:** indicate the date and the user of the last modification

__last_update: information not actually stored in the database, used for concurrency control

Tasks

Add the following fields to the **Student** model:

Date of birth (Date type)

Option: string (character field)

Year: integer (e.g., first year, second year, etc.)

Relational Fields

In Odoo, there are two types of fields:

Simple fields (boolean, integer, date, float, char, text, selection, binary, etc.)

Relational fields (Figure 28).

many2one(obj, ondelete='set null'...) Relation à un objet parent (en utilisant une clé étrangère)	• obj : _name (nom) de l'objet de destination (**obligatoire**) • ondelete : manipulation de suppression, par exemple *'set null'*, *'cascade'*;
one2many(obj, field_id...) Relation virtuelle vers plusieurs objets (inverse de many2one)	• obj : _name (nom) de l'objet de destination (**obligatoire**) • field_id : nom du champ many2one inverse, c'est-à-dire : clé étrangère correspondante (**obligatoire**)
many2many(obj, rel, field1, field2...) Relation bidirectionnelle entre plusieurs objets	• obj : _name (nom) de l'objet de destination (**obligatoire**) • rel : nom optionnel de la table de relation à utiliser (par défaut : autoattribué sur la base des noms de modèles) • field1 : nom du champ dans la table rel stockant l'id de l'objet courant (par défaut : en fonction du modèle) • field2 : nom du champ de la table rel stockant l'id de l'objet cible (par défaut : en fonction du modèle)

Figure 28 Relational fields

Example:

The following example shows the declaration of two classes containing both simple fields and relational fields, such as the field **student_id**, which is of type **Many2one** (Figure 29).

```
class schoolresults_detail(models.Model):
    _name = "schoolresults.detail"
    _description = "Student's student secondary education results."

    student_id = fields.Many2one("student.student", "Student", ondelete="cascade")
    subject_id = fields.Many2one("schoolresults.subject", "Subject")
    result = fields.Float("Result")
```

```
class student_student(models.Model):
    _name = "student.student"
    _description= "Student Information"

    name = fields.Char(string="Name", required=True, index=True, translate=True)
    active = fields.Boolean(string="Active", default=True)
    image = fields.Binary("Image")
    uni_no = fields.Char(string="Ministry University No.", required=True, copy=False)
    seat_no = fields.Char("Seat No.", copy=False)
    dob = fields.Date(string="Date of Birth", required=True)
    age = fields.Integer(string="Age")
    gender = fields.Selection([("male","Male"),("female","Female")], "Gender", default="male")
    result_ids = fields.One2many("schoolresults.detail", "student_id", "School Results")
    hobbies_ids = fields.Many2many("hobbies.detail", "student_hobbies_rel", "student_id","hobbie_id", "Hobbies Information")
```

Figure 29 Definition of models with relational fields

Tasks

Create a **Cart** model, a **Customer** model, and a **Product** model. The logic is as follows:

-A cart can contain one or more products.

-A cart belongs to one customer.

7.5.3 The Presentation Layer in Odoo – Views, Menus, and Actions

Objective

Views are defined using XML files; they determine how records are displayed. The objective of this exercise is to manage the display of the models created in the previous exercise (Figure 30).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<openerp>
   <data>
        [views definitions]
        [actions definitions]
        [menus definitions]
   </data>
</openerp>
```

Figure 30 XML file containing the declaration of a view

As you can see, the XML file in which you define the design and presentation aspects of your Odoo module must include three main definitions: views, actions, and menus.

Views

Below is an example of a basic view structure (Figure 31):

```xml
<record model="ir.ui.view" id="view_id">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <!-- view content: <form>, <tree>, <graph>, ... -->
    </field>
</record>
```

Figure 31 Declaration of a view in an XML file

There are several types of views; in this section, we will focus on **Tree Views** and **Form Views**.

Tree Views

Let us implement a **Tree View** for our module.

Using the basic structure presented earlier, we insert the declaration block corresponding to the tree view (Figure 32).

```xml
<openerp>
    <data>


        <record model="ir.ui.view" id="produit_tree">
            <field name="name">produit.tree</field>
            <field name="model">apprenti_commercant.produit</field>
            <field name="arch" type="xml">
                <tree string="Produit list">
                    <field name="name"/>
                </tree>
            </field>
        </record>
```

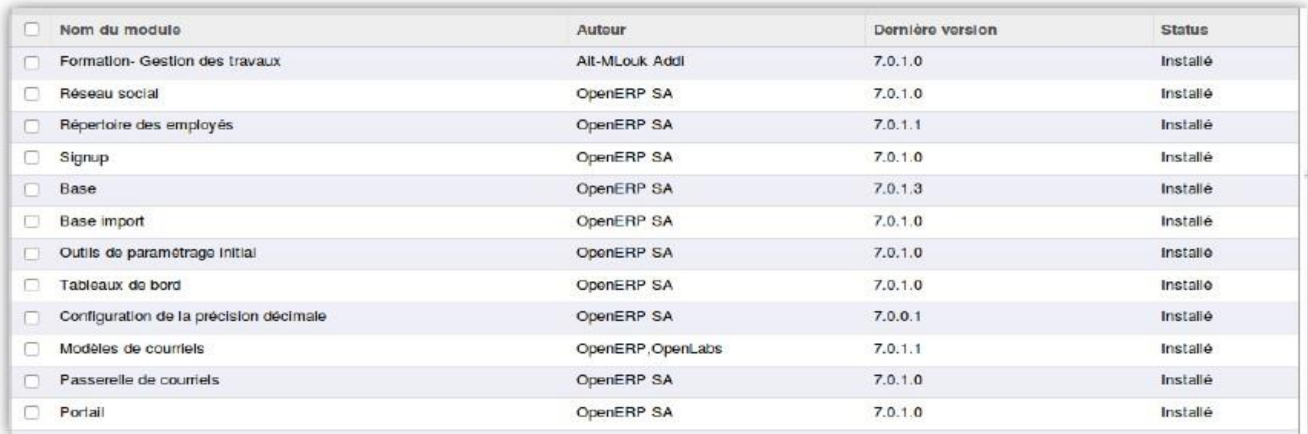
Figure 32 Tree view

Note:

Update the module by clicking on **“Settings”** at the top of the page. Then, search for your module in the list of modules, select it, and click on **“Upgrade”**.

Return to the module page and refresh it; you should obtain the following result (Figure 33).

Figure 33 Display of a Tree View



Nom du module	Autour	Dernière version	Status
Formation- Gestion des travaux	Ait-MLouk Addi	7.0.1.0	Installé
Réseau social	OpenERP SA	7.0.1.0	Installé
Répertoire des employés	OpenERP SA	7.0.1.1	Installé
Signup	OpenERP SA	7.0.1.0	Installé
Base	OpenERP SA	7.0.1.3	Installé
Base import	OpenERP SA	7.0.1.0	Installé
Outils de paramétrage initial	OpenERP SA	7.0.1.0	Installé
Tableaux de bord	OpenERP SA	7.0.1.0	Installé
Configuration de la précision décimale	OpenERP SA	7.0.0.1	Installé
Modèles de courriels	OpenERP,OpenLabs	7.0.1.1	Installé
Passerelle de courriels	OpenERP SA	7.0.1.0	Installé
Portail	OpenERP SA	7.0.1.0	Installé

Figure 34 Display of a List View

In Figure 34, you can see an example of a display using a list view.

Form Views

Let us now implement a **Form View** for our module.

Using the same basic structure, we insert the declaration block corresponding to the form view. (Figure 35)

```
<record model="ir.ui.view" id="produit_form">
    <field name="name">produit.form</field>
    <field name="model">apprenti_commercant.produit</field>
    <field name="arch" type="xml">
        <form string="Produit Form">
            <sheet>
                <group>
                    <field name="name"/>
                </group>
            </sheet>
        </form>
    </field>
</record>

<!-- window action -->
<!--
    The following tag is an action definition for a "window action",
    that is an action opening a view or a set of views
-->

<record model="ir.actions.act_window" id="produit_list_action">
    <field name="name">Produits</field>
    <field name="res_model">apprenti_commercant.produit</field>
    <field name="view_type">form</field>
    <field name="view_mode">tree,form</field>
    <field name="help" type="html">
```

Figure 35 Form view associated with our model

Below is an example of a Form View display (Figure 36).

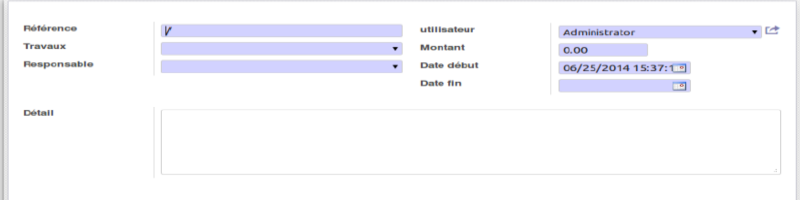


Figure 36 Display of a Form View

Actions

The XML declaration of an action is as follows (Figure 37):

model: the table (database) used to store actions

id: the identifier of the action in the *ir.actions.act_window* table; it must be unique

name: the name of the action (required)

res_model: the model (Python class) associated with the view (required)

```
<record model="ir.actions.act_window" id="work_action_work">
        <field name="name">Works</field>
        <field name="res_model">formation.work</field>
        <field name="view_mode">tree,form</field>
</record>
```

Figure 37 Example of an action declaration

view_mode: the list of modes available for displaying records Bas du formulaire

Menus

The XML declaration of a **main menu (level 1)** is defined as follows:

```
<menuitemid="work_menu_root"name="Travaux"/>
```

The declaration of a **level 1.1 menu** (without action = not clickable) is defined as follows:

```
<menuitemid="work_menu"parent="work_menu_root"name="Travaux"/>
```

The declaration of a **level 1.1.1 menu** (clickable menu with action) is defined as follows:

```
<menuitemid="work_work_menu"parent="work_menu"name="Travaux"action="[Action_ID]"/>
```

Before using clickable menus, it is necessary to define the action using: action="[ACTION_ID]", which will be triggered when the menu is selected.

Example 1

Assuming that the actions are already defined, create the following menus and submenus (Figure 38).

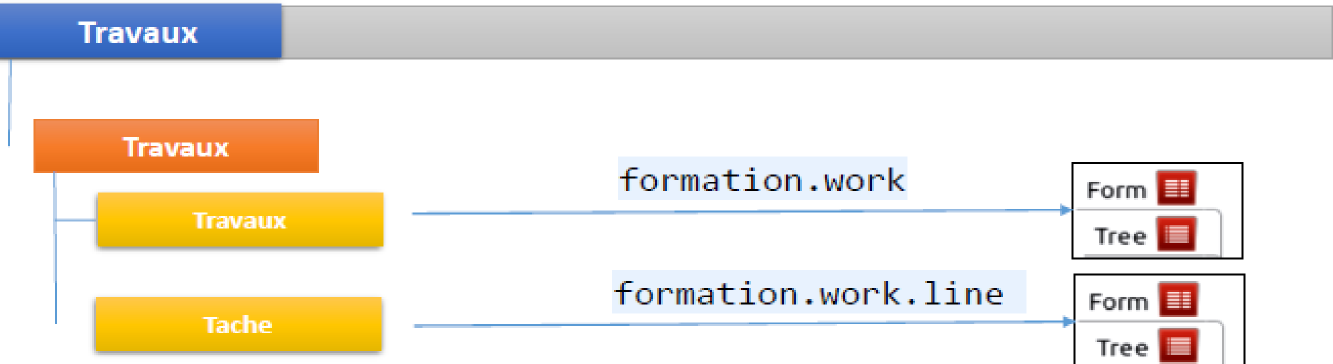


Figure 38 Example of menus and submenus

Example 2

1. Create a **Cart** model, a **Customer** model, and a **Product** model. The logic is as follows:

-A cart can contain one or more products.

-A customer can own one or more carts.

2. Create the corresponding views, knowing that:

-The **Cart** model has the following attributes: name, product_ids, and client_id

-The **Customer** model has the following attributes: name, lastname, and personality

-The **Product** model has the following attributes: name and product_id

7.5.4 Inheritance in Odoo

Objective

The concept of inheritance is a powerful mechanism that allows both the reuse and enhancement of existing code and modules, while preserving the original implementation.

Inheritance, which is a fundamental concept of object-oriented programming, is widely used in Odoo in various forms, including the extension of Python classes, inheritance of views and interfaces, and inheritance of web pages, among others.

The objective in Odoo is to reuse existing modules, classes, or views and adapt them to meet the specific requirements of the module you intend to develop for an organization.

Model Inheritance

```
#inheritance
class resPartner(models.Model):
    _inherit = 'res.partner'

    national_id = fields.Char(string='National ID')
```

Figure 39 Model inheritance

By using the _inherit attribute, you can inherit an existing model and extend it by adding new fields or methods. In this case, you continue working on the same database table (Figure 39).

If you also add the _name attribute, Odoo will create a new table that includes the fields and

```
#inheritance
class resPartner(models.Model):
    _name='responsable'
    _inherit = 'res.partner'

    national_id = fields.Char(string='National ID')
```

Figure 40 Model inheritance and creation of a new class

methods of the inherited model (Figure 40).

View Inheritance

Just as a model can inherit from an existing one, the same principle applies to views.

You can reuse an existing view to define a new one (Figure 41).

```
<!-- View inheritance for Partner Model-->

<record id="view_partner_form_inherited_national_id" model="ir.ui.view">
    <field name="name">Inherited Form View for National ID field</field>
    <field name="model">res.partner</field>
    <field name="inherit_id" ref="base.view_partner_form"/>
    <field name="arch" type="xml">
            <xpath expr="//field[@name='mobile']" position="attributes"> -->
                <attribute name="invisible">1</attribute> -->
            </xpath> -->
            <xpath expr="//field[@name='name']" position="after"> -->
                <field name='national_id'/> -->
            </xpath> -->
    </field>
</record>
```

Figure 41 View inheritance

To do this, simply assign the value **inherit_id** to the appropriate field and specify the ID of the view you want to inherit from.

The same principle applies to both **Form Views** and **List Views**. The main difference lies in the use of the **inherit_id** field, which identifies the view from which the new view will inherit.

Tasks

Choose one of the existing modules in the *addons* directory (developed by the Odoo community) and try to inherit it within a new module that you create.

Note:

In the previous exercises, we examined the main files and directories that make up an Odoo module. However, there are others, as shown in the figure below (Figure 42).

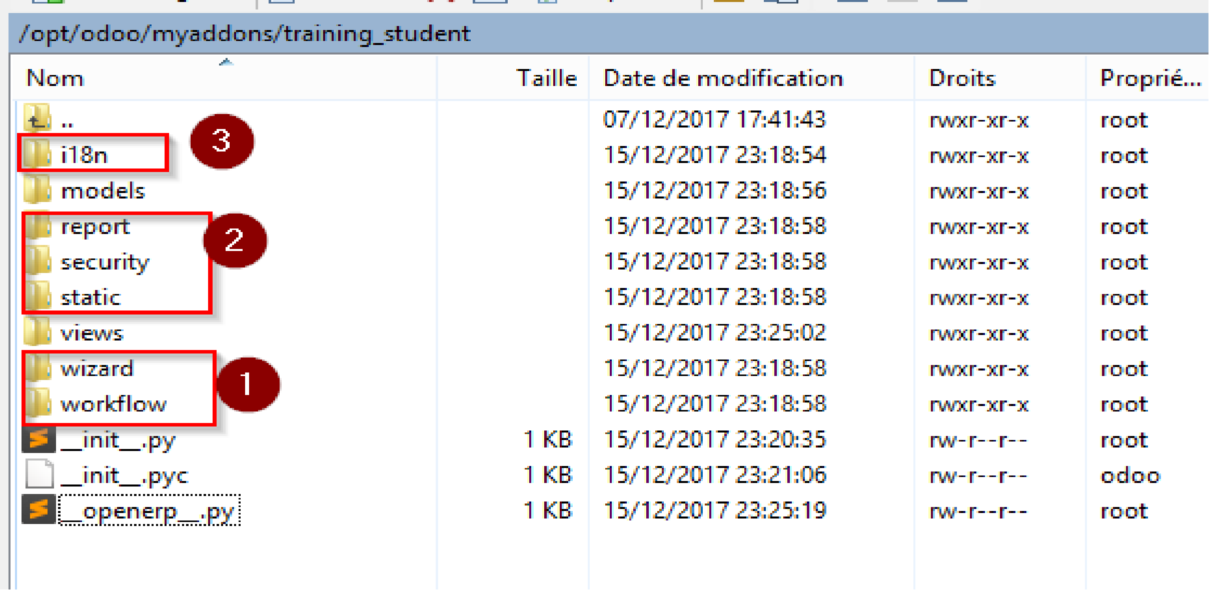


Figure 42 Structure of an Odoo module

1 → **Wizard:** used for assistant interfaces (e.g., performing calculations)
Workflow: used to define process steps or business workflows

2 → **Report:** used for generating PDF reports

Security: used to manage access rights

Static: used for storing logos, images, and other static assets

3 → **i18n:** used for translation and internationalization

8. Conclusion

This chapter has provided a comprehensive overview of Enterprise Resource Planning (ERP) systems and their fundamental role in modern industrial and organizational environments. As businesses face increasing complexity and the need for real-time decision-making, ERP systems emerge as essential tools that enable the integration, automation, and optimization of core business processes. By centralizing data within a unified system, ERPs ensure consistency, improve coordination between departments, and enhance overall operational efficiency.

Through the exploration of ERP principles, functionalities, and key characteristics, it becomes evident that these systems are not only technological solutions but also strategic assets that

support organizational performance and competitiveness. The distinction between proprietary, open-source, and cloud-based ERP systems highlights the diversity of available solutions, allowing organizations to choose systems that best align with their technical capabilities, financial resources, and business objectives.

A particular emphasis was placed on the Odoo ERP system, which stands out due to its modular architecture, flexibility, and accessibility. As an open-source platform, Odoo offers a dynamic and scalable environment that supports customization and continuous improvement. Its use of modern technologies such as Python, XML, and Object-Relational Mapping (ORM), combined with its MVC architecture, makes it a powerful yet accessible tool for both learning and professional application.

Furthermore, the practical exercises introduced throughout this chapter played a crucial role in bridging the gap between theoretical concepts and real-world implementation. By guiding learners through module creation, data modeling, interface design, and inheritance mechanisms, these exercises provide hands-on experience that strengthens technical skills and deepens understanding of ERP development processes.

In conclusion, ERP systems are key enablers of digital transformation, allowing organizations to streamline operations, enhance data visibility, and support informed decision-making. Odoo, in particular, exemplifies how modern ERP solutions can combine functionality, flexibility, and ease of use. Mastering such systems equips students and professionals with valuable competencies that are increasingly demanded in today's data-driven and technology-oriented business environment [4].

Chapter 3: Unified Modeling Language (UML)

1. Introduction

In any system design project, clear requirements, structured thinking, and effective communication among stakeholders are essential. The Unified Modeling Language (UML) has become the de facto standard to meet these needs. It provides a powerful graphical notation that is both rigorous and intuitive, allowing for the modeling of both the static and dynamic aspects of software or organizational systems.

This chapter offers a general introduction to the UML language without delving into all its elements. The goal is to provide students with the fundamental skills needed to understand, interpret, and create simple yet meaningful models. We will focus on three key diagrams that play a central role in the functional and technical specification of a system:

-**The use case diagram**, which represents the interactions between actors and the system;

-**The sequence diagram**, which describes the flow of messages in a given scenario;

-**The class diagram**, which models the core entities of the system and their relationships.

"UML is not just a modeling notation; it is a language for thinking about and communicating designs."[5]

2. Definitions of UML:

Definition 1: The Unified Modeling Language (UML) is a graphical modeling language based on pictograms, designed as a standardized method for visualizing systems, primarily in the fields of software development and object-oriented design.

UML is the result of a synthesis of earlier object-oriented modeling languages, namely Booch, OMT, and OOSE. It originates primarily from the work of Grady Booch, James Rumbaugh, and Ivar Jacobson.Today, UML is a standard officially adopted by the Object Management Group (OMG).

Historical Overview: UML 1.0 was standardized in **January 1997**, and UML 2.0 was adopted by the **Object Management Group (OMG)** in **July 2005**.The latest version of the specification officially approved by the OMG is **UML 2.5.1**, released in **2017**.

Definition 2: The Unified Modeling Language (UML) is a graphical modeling language used to specify, visualize, construct, and document the artifacts of a software system, particularly in the context of object-oriented development. It provides a standardized set of notations to represent both the structural and behavioral aspects of systems.

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [6].

3. UML Diagrams:

UML diagrams are divided into two main categories:

-Structural diagrams, which represent the static view of the system;

-Behavioral diagrams, which illustrate the dynamic view of the system.

There are a total of **13 UML diagram types (Figure 43)**, but in this course, we will focus on **three key diagrams**:

-The **use case diagram**, which models the system's behavior and functionalities from the user's perspective;

-The **class diagram**, which describes the static structure of the system, including object types and their relationships;

-The **sequence diagram**, which depicts the chronological interactions between objects and actors.

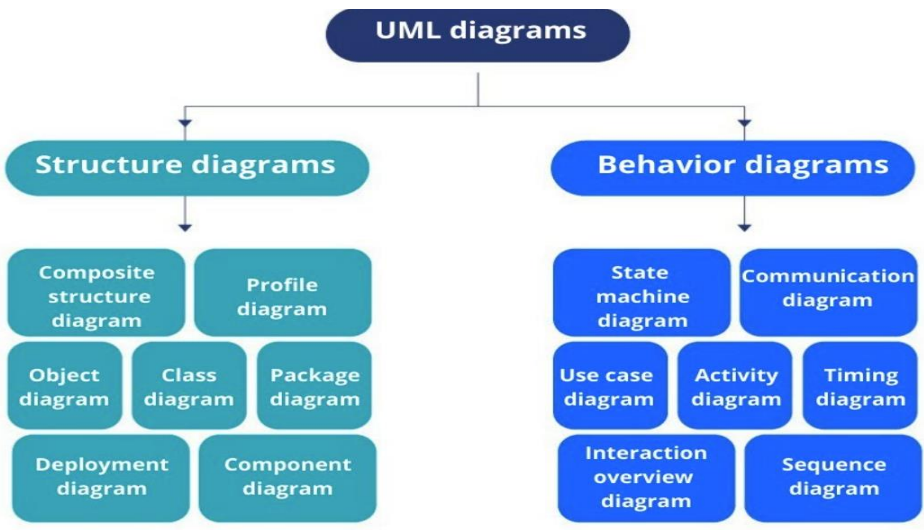


Figure 43 UML diagrams structural and behavioral

4. Use Case Diagram

The use case diagram is one of the most commonly used diagrams in UML, especially during the early stages of functional system modeling. It is used to represent **the services or functionalities provided by a system to its users**, known as **actors.** These actors can be individuals, external systems, or any other entities interacting with the system.

The primary goal of this diagram is to **describe the system from an external viewpoint**, focusing on **functional requirements** rather than technical implementation. It serves as an effective communication tool among stakeholders (clients, end-users, analysts, developers), as it is intuitive and easy to understand even for non-technical audiences.

A use case diagram typically consists of:

-**Actors**: depicted as stick figures, representing entities that interact with the system.

-**Use cases**: shown as ellipses, representing the system's functionalities or usage scenarios.

-Relationships: arrows or lines between actors and use cases (and sometimes between use cases), indicating interactions, inclusions, extensions, or generalizations.

For example, in a library management system, common use cases might include Borrow Book, View Catalog, and Return Book, with actors such as Reader, Librarian, or even an External Payment System.

The use case diagram answers the question: **What does the system do?** or more precisely: **What interactions are possible between the users and the system?**

4.1 Main Elements of a Use Case Diagram

A use case diagram is mainly composed of two key elements: **actors** and **use cases (Figure 44).**

Actor→An actor represents an **external entity** that interacts with the system. This may be: a **person** (user, an administrator), **another software system** (a payment service), or a **physical object** (a sensor or card reader).

The actor defines a **role**, not a specific individual: the same entity can play multiple roles.It is identified by **the name of the role**, not by the actor's personal name.

Use Case→ A use case refers to a **functionality or service provided by the system and visible from the outside**.It is an **action initiated by an actor**, to which the system responds. It is typically named using an **infinitive verb**, such as “Check balance”, “Manage account”, or “Confirm order”.

These elements are connected through **interaction relationships**, showing who does what with the system.

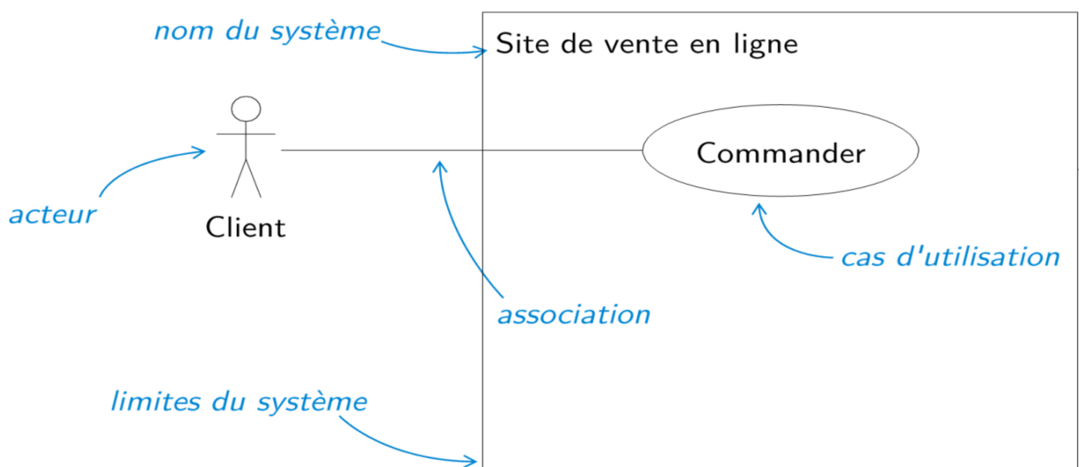


Figure 44 Main components of a use case diagram

Association in a Use Case Diagram →An **association** is a **fundamental relationship** in a use case diagram. It connects an **actor** to a **use case**, indicating that the actor **interacts with the system** by initiating or participating in that use case.

-It is shown as a **simple line** between the actor and the use case.

-It expresses that the actor **can trigger or benefit from** the functionality described.

-One **actor** may be associated with **multiple use cases**, and a single **use case** may involve **multiple actors**. **(Figure 45)**

The association does not specify the detailed behavior but clearly indicates the **potential for an actor to make use of a system function**.

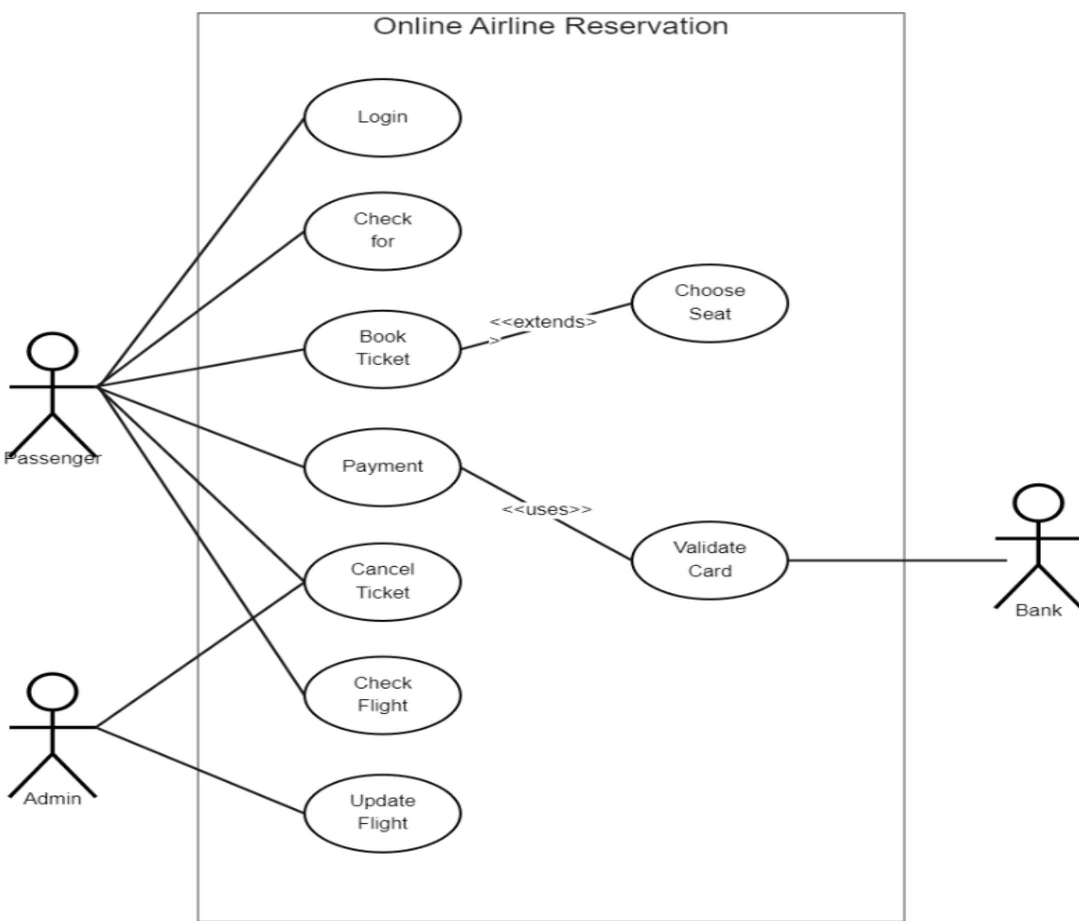


Figure 45 Example of a use case diagram

In UML use case diagrams, **role generalization** is used when one actor (the child) inherits the behavior and relationships of another actor (the parent). This is useful when multiple roles share common interactions with the system but also have specific behaviors.

As shown in the figure "**Figure 46**", both Supervisor and Student are considered users above all.

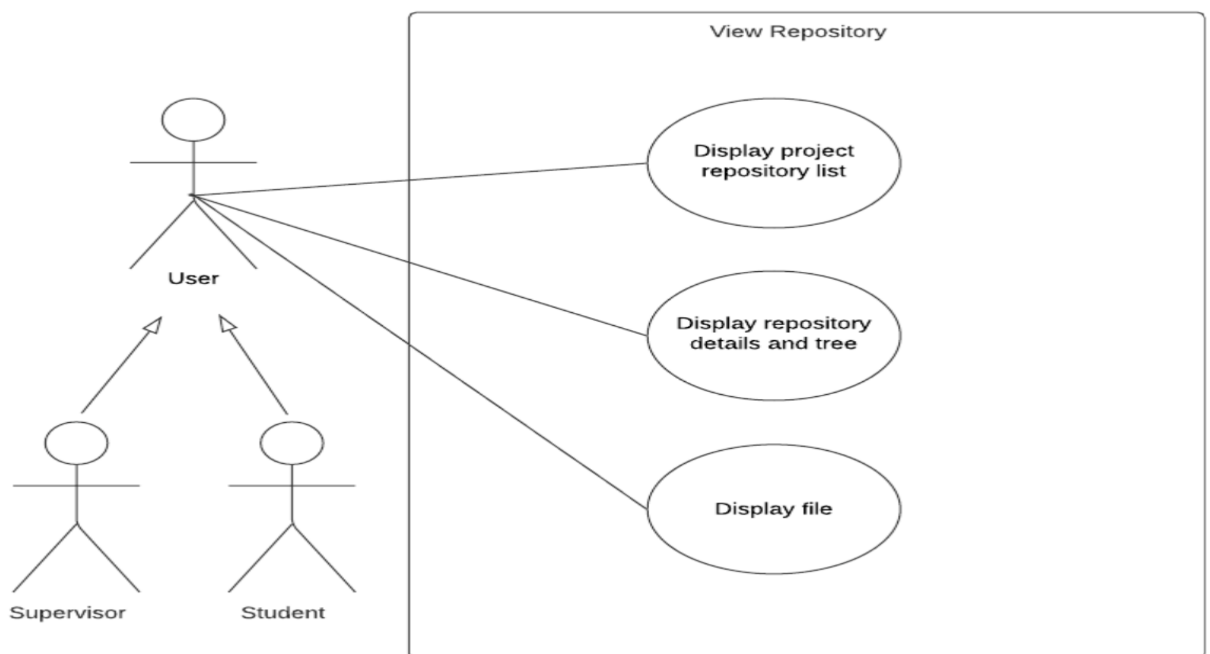


Figure 46 Role generalization

In a use case diagram, two common relationships can exist between use cases: «include» and «extend». The «include» relationship represents a **mandatory inclusion** of common behavior (**Figure 47**): it allows one use case to incorporate the functionality of another use case that is shared across multiple scenarios. This helps to reduce redundancy and improve modularity the included use case is always executed when the base use case runs. For example, both Place Order and Cancel Order might include Authenticate User. On the other hand, the «extend»relationship represents an **optional or conditional extension** of behavior (**Figure 48**): it allows a use case to insert additional steps into another use case under specific conditions. The extended use case defines a base scenario, while the extending use case adds alternative or exceptional behavior. For instance, Make Payment might be extended by Apply Discount only if the user is eligible. These two relationships help model reusable and adaptable functionality in complex systems.

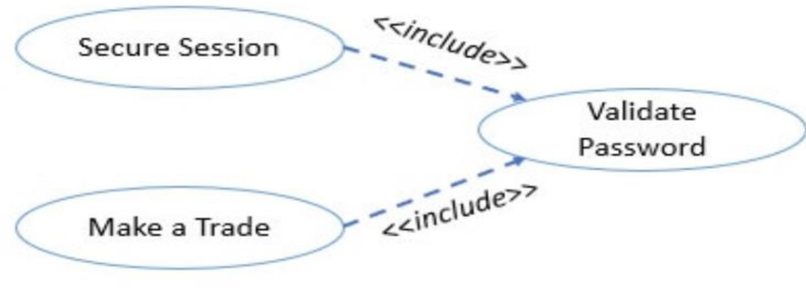


Figure 47 Include use case Relationship

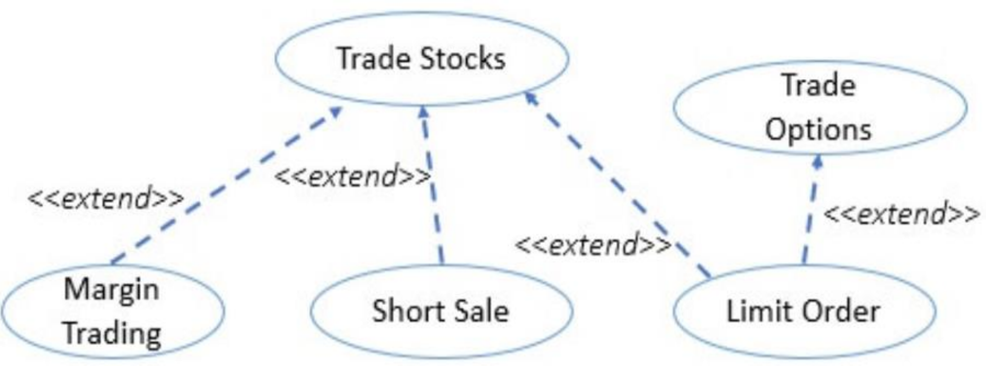


Figure 48 Extend use case Relationship

Guidelines for Designing a Use Case Diagram

To ensure clarity and usefulness, the following best practices are recommended when designing a use case diagram:

-**Limit the number of use cases** to around **6 to 8 per diagram**. More than that can reduce readability and make the diagram harder to interpret.

-**Break down the system** into subsystems or logical functionalities if needed, and use **multiple diagrams** to cover different areas.

-**Avoid excessive use of relationships** between use cases (such as include or extend): use them only when they add real value, to avoid cluttering the diagram.

-For **detailed behavior**, prefer **textual descriptions** such as structured use case narratives or detailed scenarios.

-**Dynamic scenarios** involving interactions between actors and system elements can be illustrated using **sequence diagrams**, which show the temporal flow of messages.

5. Class Diagram

Object-Oriented Design: The system is represented as a set of interacting objects.

The class diagram→ Describes the internal structure of the software; defines classes, their attributes, methods, and relationships.

Object Diagram→Represents the state of the software at a specific moment (objects and their relationships); evolves during the execution of the software:

-Creation and deletion of objects.

-Changes in object state (attribute values).

-Changes in relationships between objects.

The **class diagram** is one of the core elements of UML modeling. Its primary goal is to **show the static structure of the system**, by representing the different **classes**, along with their **attributes** (data) and **methods** (functions).

This diagram is **mainly intended for developers**, as it provides a comprehensive view of the software architecture, object types, and the relationships between them (associations, compositions, generalizations, etc.).It often serves as a **starting point for implementation** in an object-oriented programming language.

In summary, a class diagram allows the representation of:

-the **classes** in the system;

-their **attributes** (properties or characteristics);

-their **methods** (behaviors or operations);

-and the **relationships** between classes (such as association, inheritance, or dependency).

Definition of an Object: In object-oriented modeling, an **object** is a **concrete or abstract entity** from the **application domain** being modeled. **(Figure 49)**

It can represent a real-world element (e.g., a customer, a product, an order) or a conceptual one (e.g., a transaction, an event, a business rule).

An object is defined by three fundamental characteristics:

Identity: what distinguishes it from other objects, even if they have the same attribute values (technically, its memory address).

State: the set of **attribute values** it holds at a given moment.

Behavior: the set of **operations** or **methods** it can perform.

Thus, an object is both a **data unit** and a **processing unit**, capable of interacting with other objects within the system.

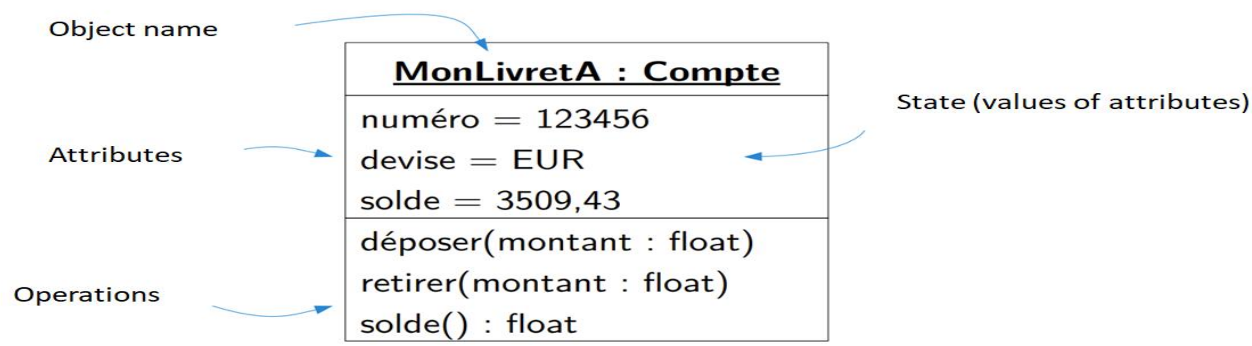


Figure 49 Declaration of an object

Class and Object: In object-oriented modeling, a **class** is a **template or blueprint** used to create objects. It represents a **group of objects with the same nature**, meaning objects that share: the **same attributes** (properties or data), and the **same operations** (methods or behaviors).

In other words, a class defines the **common structure and behavior** of all its instances. An **object** is a **concrete instance of a class**.

When an object is created from a class, we say the class is **instantiated**. Each object has its **own values for the attributes**, although it follows the structure and behavior defined by the class.

Example 1: The class Car might define attributes such as color, brand, maxSpeed, and methods like accelerate() or brake().The object myCar is an instance of this class with color = red, brand = Toyota, etc.

Example 2: Three objects MonCompteLivretA, MonCompteJoint, and MonCompteSuisse are instances of the class Compte. **(Figure 50)**

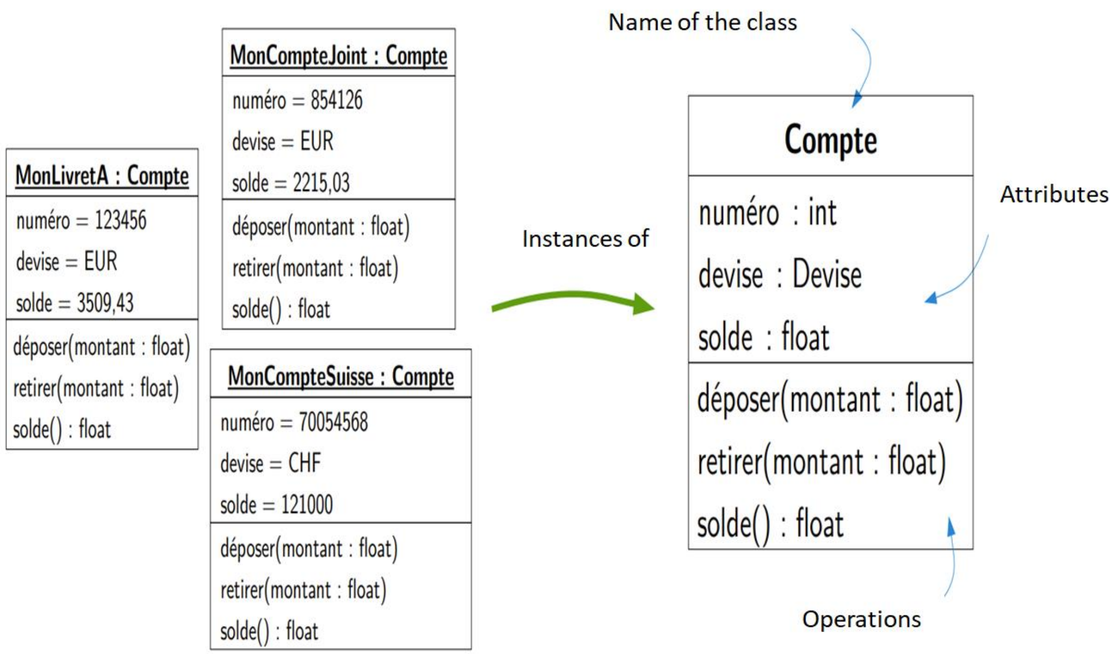


Figure 50 Declaration of three objects instances of a class

Attributes in UML Class Diagrams: In UML, **attributes** represent the **properties or characteristics** shared by all instances of a class. Each object holds a **specific value** for each attribute, but the **definition of the attribute** its name and type is specified at the class level. Attributes are always associated with a **data type**, which can be:

-a **primitive type** (e.g., int, bool, float); a **complex type** (such as a Date object); or an **enumeration**.

An **enumeration** is a special type of class that defines a fixed set of constant values. It is identified by the stereotype <<enumeration>> and consists only of named values (literals). In

class diagrams, an attribute can be declared with an enumeration as its data type, meaning that it can only take one of the predefined values. Enumerations provide a clear and constrained way to model categorical or status-type data within a system.

Object State and Attribute Values: The **state of an object** is defined by the **values of its attributes** at a given moment. While all objects of the same class share the **same structure** (the same set of attributes), each object can hold **different values** for those attributes. This is what allows objects to behave uniquely within the same class definition.

It is also possible for two distinct objects each with its own **identity** (memory address) to have **identical attribute values**, yet they remain different instances (**Figure 51**). The identity of an object is independent of its state, meaning that object uniqueness is preserved even if two objects temporarily look the same.

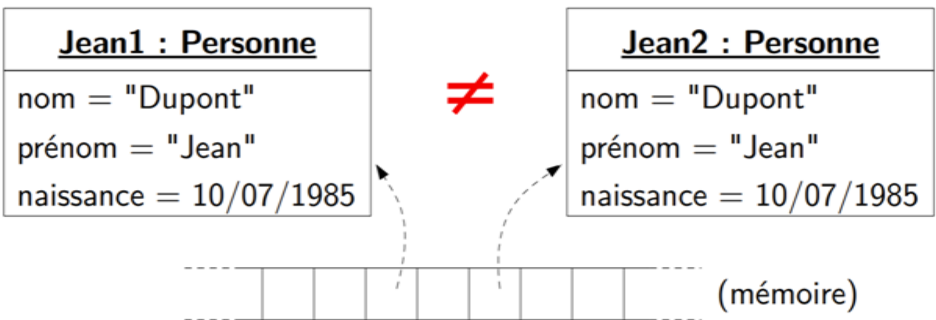


Figure 51 Different objects with the same values of attributes

In UML terminology, an **operation** refers to the **specification or declaration** of a behavior that instances of a class can perform (**Figure 52**). It defines the **signature** that is, the method name, parameters, and return type without describing how the behavior is implemented.

The **implementation** of that operation, written in code within a specific programming language, is what is commonly called a **method**. However, this dual usage of the term “method” creates a **semantic ambiguity**:

-In modeling (UML), “method” is often used as a synonym for **operation**,

-In programming, “method” usually refers to the **code implementation** of that operation.

Therefore, when teaching or documenting object-oriented design, it's important to clarify the context in which the word "method" is used whether it refers to the **model-level declaration** (operation) or the **code-level definition** (implementation).

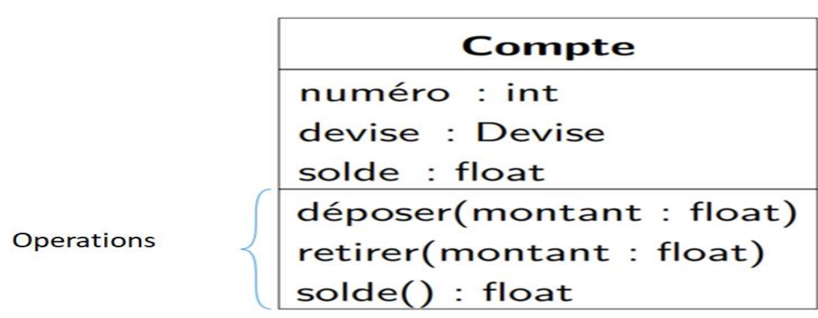


Figure 52 Operations in a class

Relationships Between Objects: In UML, a **relationship between objects** represents a **link**, typically **binary**, that connects two instances of classes (**Figure 53**). This link indicates a form of interaction or dependency between the objects during the system's execution. In most cases, there is **at most one link** between any given pair of objects for a specific **association**.

Such object-level relationships are instances of **class-level associations**, meaning they are concrete manifestations of the abstract relationships defined in class diagrams. These links are dynamic and reflect how objects **collaborate** or **communicate** at runtime.

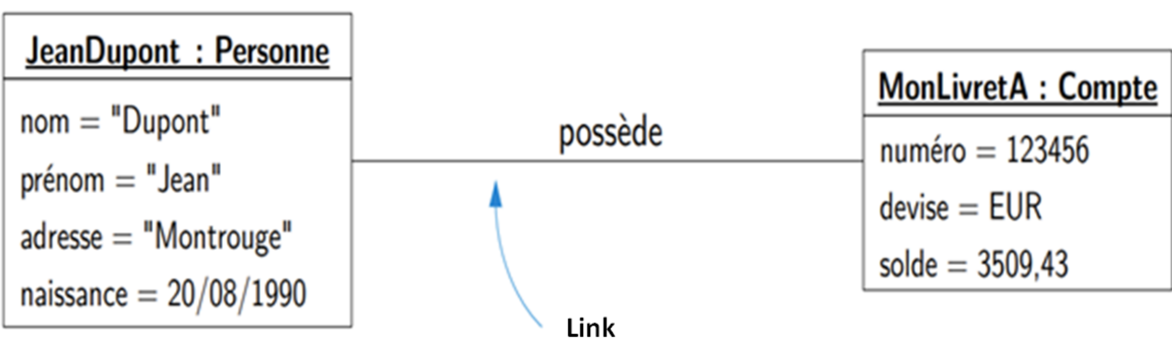


Figure 53 Relationship between objects

<u>Relationships Between Classes</u>: In UML class diagrams, a **relationship between classes** is typically modeled as a **binary association**, meaning it connects two classes **(Figure 54)**. This association represents a potential link between instances of these classes at runtime.

Each end of an association can be given a **role name**, which identifies the perspective or function of the related class in the context of the association. This role name is useful for **navigating** the relationship and for **clarifying the semantics** of the connection.

Associations also include **multiplicity constraints**, which define how many instances of one class can be associated with a single instance of the other class. For example, a "1..*" multiplicity indicates that one object must be associated with one or more objects of the other class. These constraints are essential for accurately modeling real-world cardinalities and system rules.

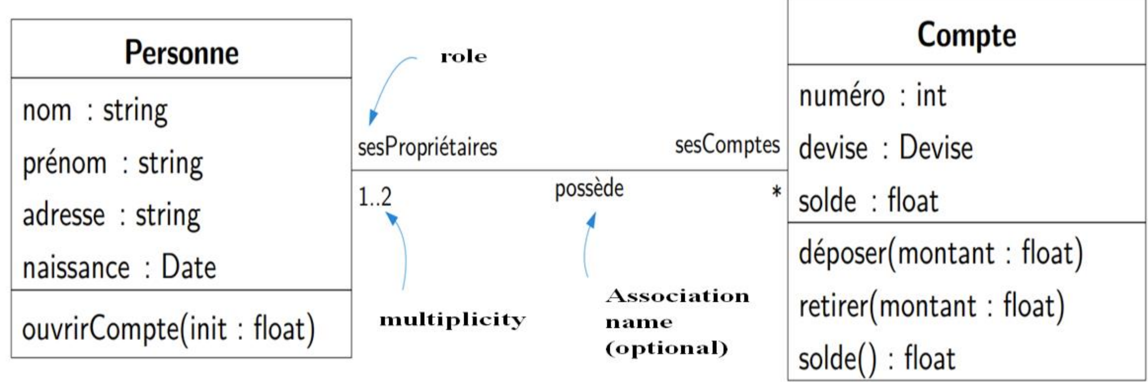


Figure 54 Relationship between classes

<u>Attributes and Associations: Initial Modeling Guidelines</u>

As a starting point in class diagram modeling, we will **limit attribute types** to **simple**, **primitive**, or **enumerated types** (e.g., int, bool, String, or predefined enumerations). This simplifies the design and focuses attention on the essential structure of the system. **(Figure 55)**

At this stage, we **avoid using attributes whose type is another class** from the same diagram. Relationships between classes especially when one class references another should be modeled explicitly through **associations**, rather than as class-type attributes. This distinction

helps maintain clarity between structural links (**associations**) and internal data (**attributes**), which is fundamental in object-oriented modeling.

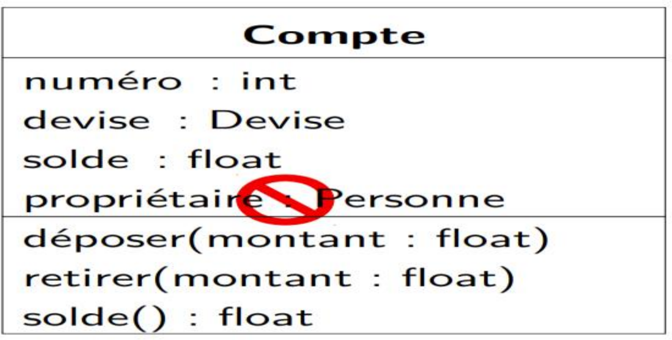


Figure 55 No instance attributes in the class in the first time

Instead of adding attributes, it is preferable to create an association to this class (Figure 56).

This approach promotes better modularity and reflects the relationship between classes more clearly in the design.

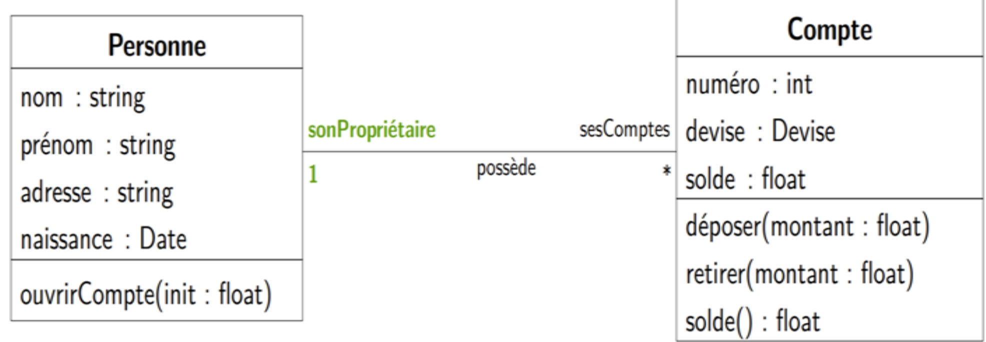


Figure 56 Association between two classes

<u>**Multiplicities in UML Associations**</u>**: Multiplicity** defines how many instances of one class can be associated with a single instance of another class in a UML class diagram. For example, if class **A** is associated with class **B**, the multiplicity specifies the **number of objects of class B** that can be linked to **one object of class A**.

Common multiplicity values include (**Figure 57**):

1 → exactly one object

0..1 → zero or one object

* (or 0..*) → zero or more objects

1..* → one or more objects

These constraints are placed at the ends of the association lines and help model real-world rules and constraints clearly. They are essential for understanding the **cardinality** of relationships between classes and for guiding correct object instantiation and interaction.

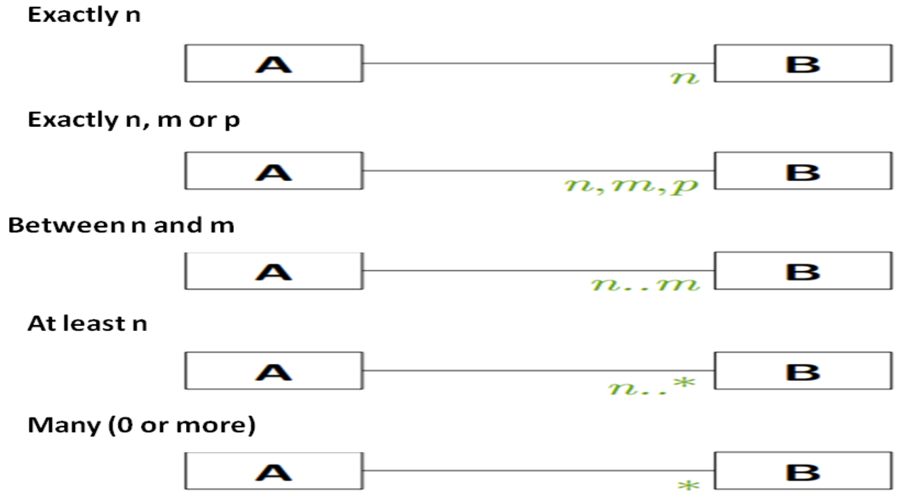


Figure 57 Multiplicities

<u>**Class Hierarchy: Generalization and Specialization**</u>**:** A **class hierarchy** is a fundamental principle in object-oriented modeling that involves **grouping classes** that share common **attributes** and **operations (Figure 58)**, and organizing them in a **tree-like structure**. This structure promotes **reusability**, **clarity**, and **scalability** in system design.

Generalization refers to the process of identifying common characteristics among several classes and abstracting them into a **superclass**. The superclass encapsulates shared features, while subclasses inherit from it **(Figure 59).**

Specialization, on the other hand, is the process of defining a **subclass** that extends or refines a more general class by adding specific attributes or behaviors.

These relationships are typically represented in UML using a solid line with a **hollow triangle** pointing toward the more general class. Hierarchies help model real-world taxonomies and promote consistent design through **inheritance**.

CompteCourant

numéro : int
devise : Devise
solde : float
découvertAutorisé : float
fraisDécouvert : float

déposer(montant : float)
retirer(montant : float)
solde() : float

CompteÉpargne

numéro : int
devise : Devise
solde : float
plafond : float
taux : float

déposer (montant : float)
retirer(montant : float)
solde() : float
calculerIntérêts() : float

Figure 58 Classes without inheritance

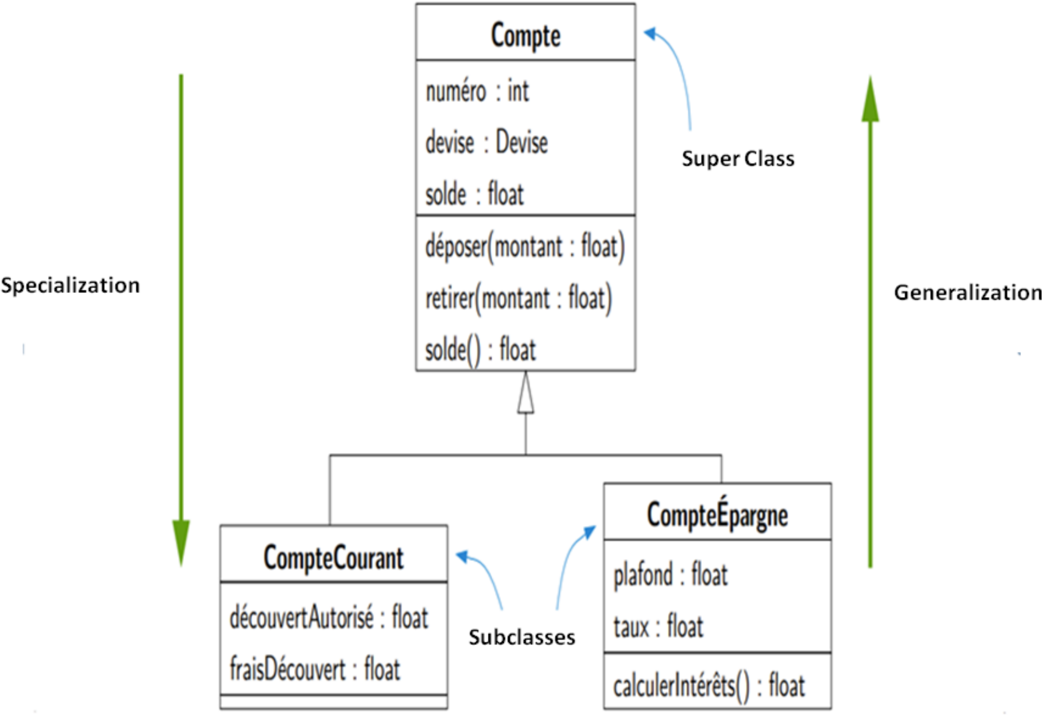


Figure 59 Classes with inheritance

Special Case: Reflexive Association in Class Diagrams: A **reflexive association** (or **self-association**) is a particular type of UML association where a **class is associated with itself**. This means that instances of the same class can be related to one another (**Figure 60**).

For example, in a class Employee, an association might indicate that **an employee can supervise other employees**. This creates a link between instances of the same class (Employee supervises Employee). In the class diagram, this is represented by a line that **loops back to the same class**, typically with role names at each end (e.g., manager and subordinate) and **multiplicities** to indicate how many instances can be related.

Reflexive associations are useful when modeling **hierarchical or networked relationships** within a single entity type.

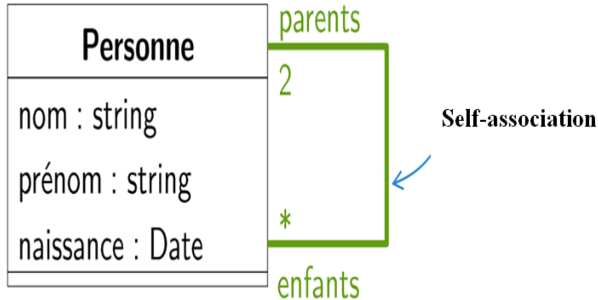


Figure 60 Self-association

Figure 61 shows an example of an object diagram associated with the previous class diagram.

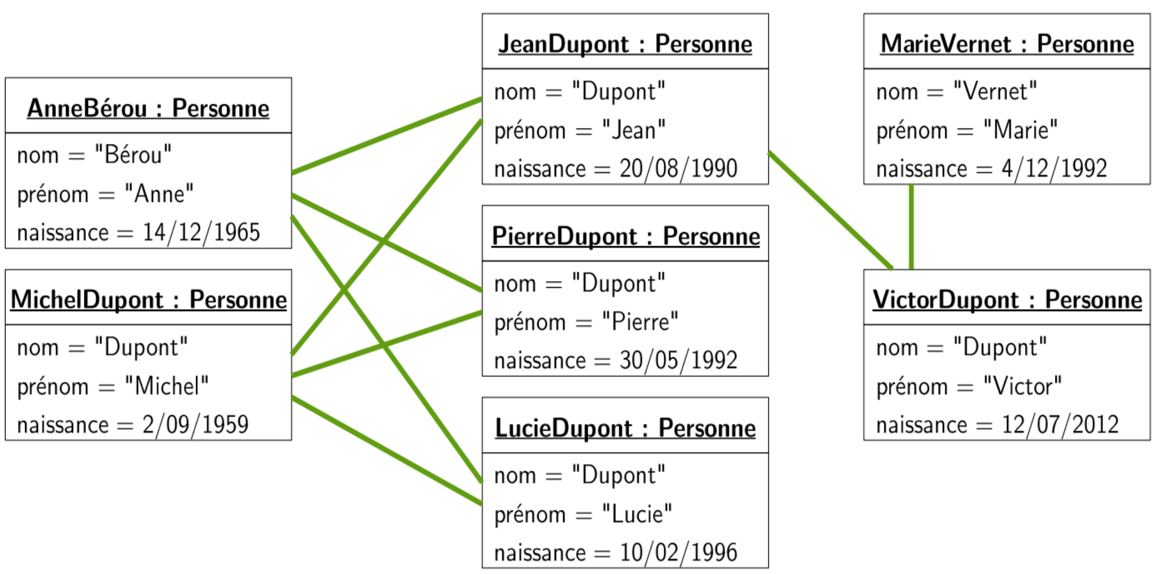


Figure 61 Example of an object diagram associated with the previous class diagram

Multiple associations: In the following class Diagram, between a ***Person (Personne)*** and an ***Apartment (Appartement)***, there may be a **tenant** relationship and an **owner** relationship **(Figure 62).**

Note: The fact that there are two associations **rents** and **offers (loue et propose)** between the two classes Person and Apartment **does not imply** that there are two relationships between every object pair (:Person and :Apartment) in the **object diagram**.

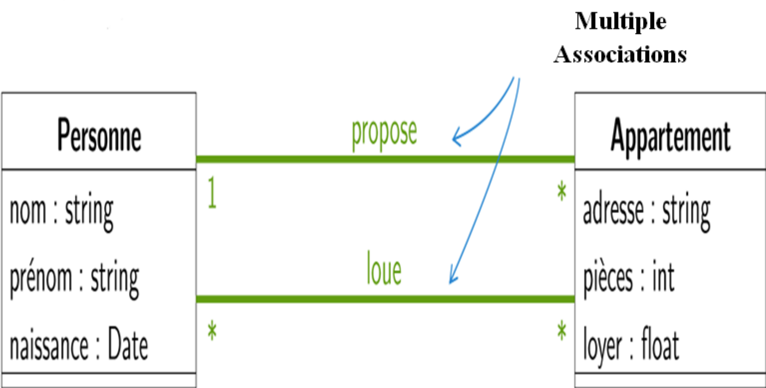


Figure 62 Example of a multiple associations

Figure 63 illustrates an example of an object diagram associated with the previous class diagram.

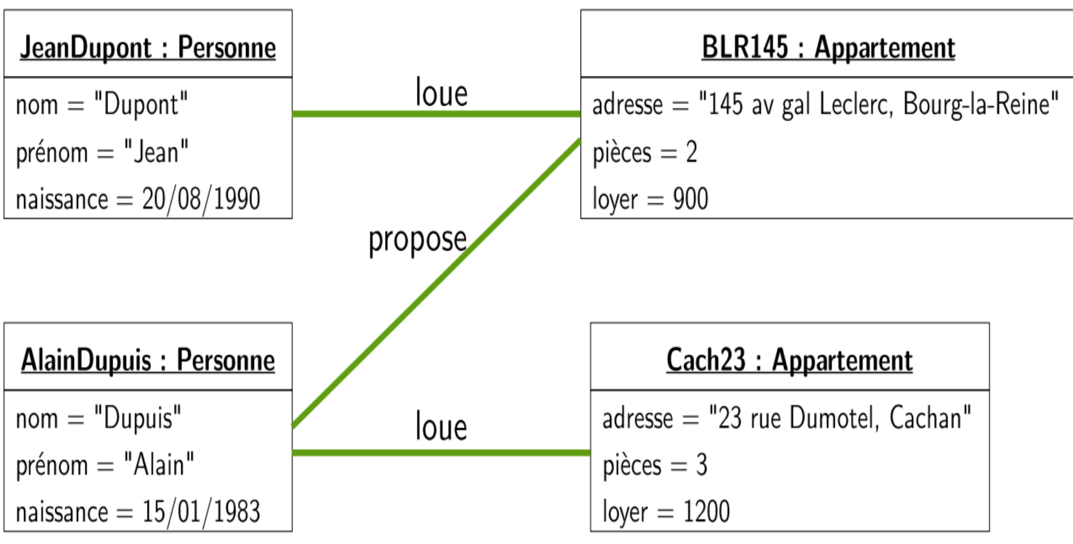


Figure 63 Example of an associate objects diagram

Association Class in UML: An **association class** is used when an association between two classes needs to carry additional information properties that do **not belong** to either of the associated classes individually but to the **relationship itself**.

For instance, consider the association Employment between the classes Company (Entreprise) and Person (Personne). This relationship may have attributes such as salary and hireDate (**Figure 64**). These details **do not naturally belong** to the Company or the Person alone, especially when a **person can have multiple jobs** within the same company or **no job at all**. Likewise, a company may offer **one or several jobs** to the same person. This situation cannot be modeled accurately with a simple association.

The solution is to introduce an **association class** a hybrid element that behaves both like an association and a class. It is connected to the association line with a **dashed line** and can contain attributes just like any other class.

Important note on multiplicity: In this context, the multiplicities (e.g., *, 1..*) **do not refer to the number of companies or persons**, but to the **number of job instances** (Employment) between a given person and company.

-A person can have **zero or many** employment relationships (*).

-A company must offer at least **one** employment relationship to a given person (1..*).

This level of expressiveness is not achievable with basic associations, which only indicate **whether or not** a relationship exists, not **how many** instances of it occur or what properties it has.

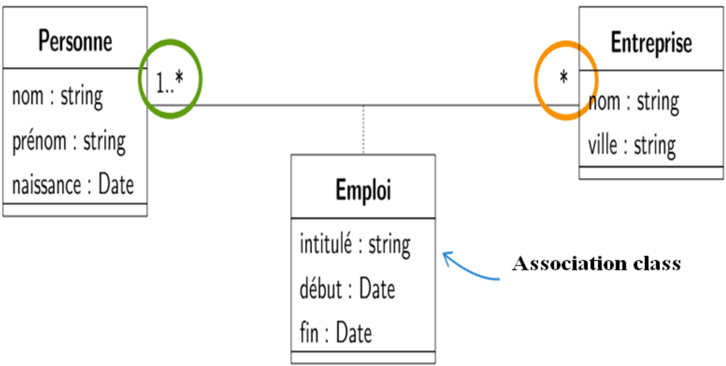


Figure 64 Association class

Figure 65 shows the equivalence in terms of classes and associations.

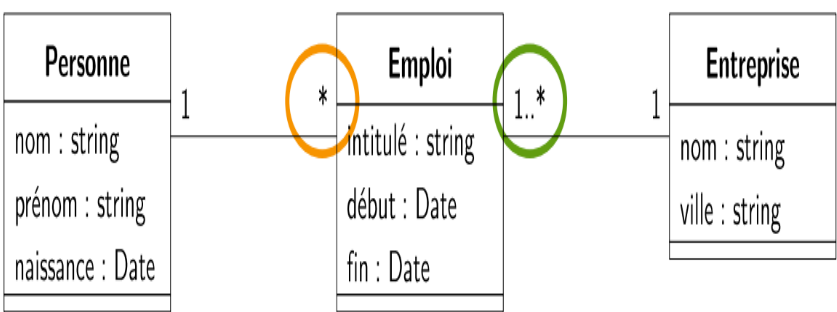


Figure 65 Associate simple classes diagram

Figure 66 illustrates an example of an object diagram (an instance of the previous class diagram).

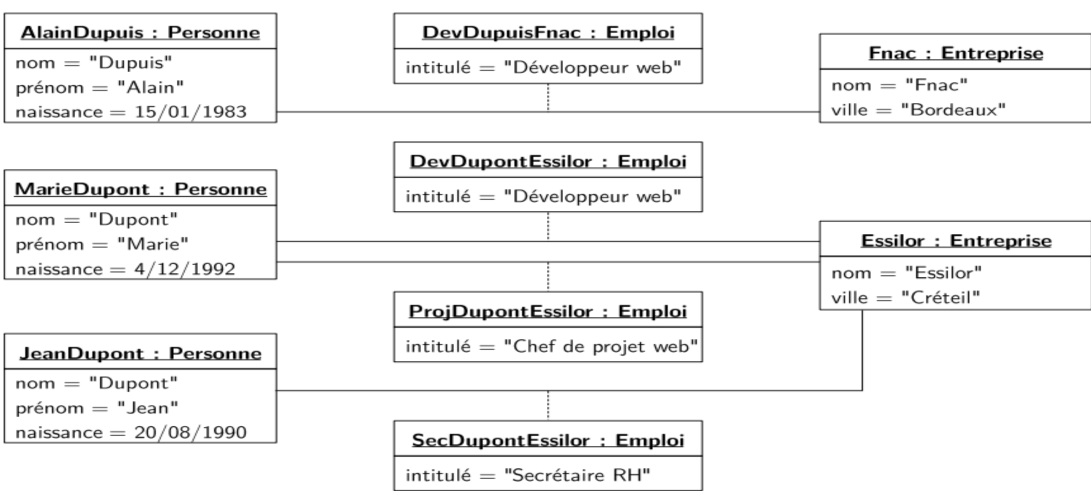


Figure 66 The associate objects diagram

Figure 67 illustrates the object diagram corresponding to the class diagram (without association classes)

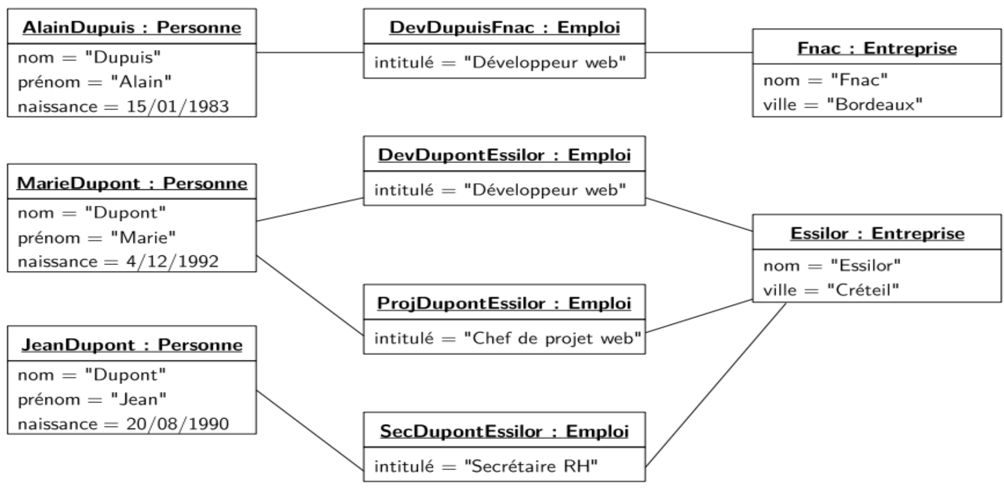


Figure 67 The associate class diagram (without association classes)

N-ary Association in UML: An **n-ary association** is a type of UML association that connects **more than two classes**. Unlike binary associations, which relate exactly two classes, an n-ary association involves **three or more participants** in a single, unified relationship.

A classic example is: “This student has this teacher for this course.”This statement involves three entities Student, Teacher, and Course which are **simultaneously** related (**Figure 68**). Representing this as three separate binary associations would not capture the precise relationship among all three roles.

In UML, an n-ary association is represented as a **diamond shape** connected to the involved classes by lines. Each line is labeled with a role name and may carry **multiplicities**.

Important note: Although n-ary associations are powerful, they are generally **discouraged unless truly necessary**, because they can be **difficult to understand, implement, and maintain**. In many cases, it is better to refactor an n-ary association into a **class with multiple binary associations** for example, by introducing a class like Assignment or Enrollment to model the interaction between student, teacher, and course.

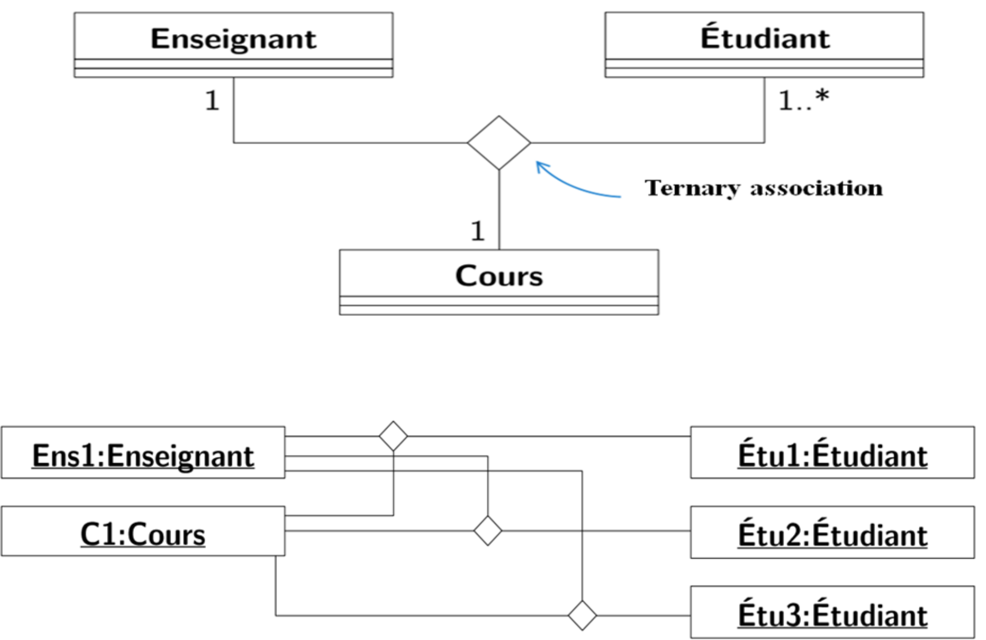


Figure 68 Example of ternary association

Aggregation in UML: Is a **special type of association** that represents a whole-part relationship between two classes. It is **asymmetric**, meaning that one class the **composite** is conceptually dominant over the other the **component**. Aggregation expresses that one object **contains** or is **composed of** others, but with varying degrees of dependency.

UML distinguishes **two types of aggregation**:

Shared Aggregation (Weak Aggregation)

This is a **loose** whole-part relationship. The component can exist **independently** of the composite. It is represented by a **hollow diamond** on the composite side. **Example***:* A Team aggregates Players. A player may belong to multiple teams or none at all and still exist independently.

Composition (Strong Aggregation)

This is a **strong** form of aggregation where the component's **lifecycle depends** on the composite. It is represented by a **filled (black) diamond** at the composite end. ***Example*****:** A House is composed of Rooms. If the house is destroyed, its rooms cease to exist as well.

6. Sequence Diagram

The **sequence diagram** is one of the **behavioral (dynamic)** diagrams in UML. It is used to represent the **interactions between system elements and external actors** over **time**.

More specifically, a sequence diagram shows **how objects and actors communicate** through **message exchanges**, organized chronologically. It is particularly effective for modeling **scenarios**, such as how a user performs a task or how different components of the system collaborate to complete an operation.

6.1 Core Elements:

Actors: External entities interacting with the system (e.g., users, external systems).

Objects: Internal elements of the system that collaborate to fulfill tasks.

Messages: Arrows representing communications (method calls, responses) between actors and objects, or between objects (**Figure 69).**

The sequence diagram provides a **graphical timeline** of how messages are exchanged either **within the system** or **between the system and its environment**. It highlights the **order of events** and the **temporal coordination** of system behavior, making it a valuable tool for understanding and specifying **system dynamics**.

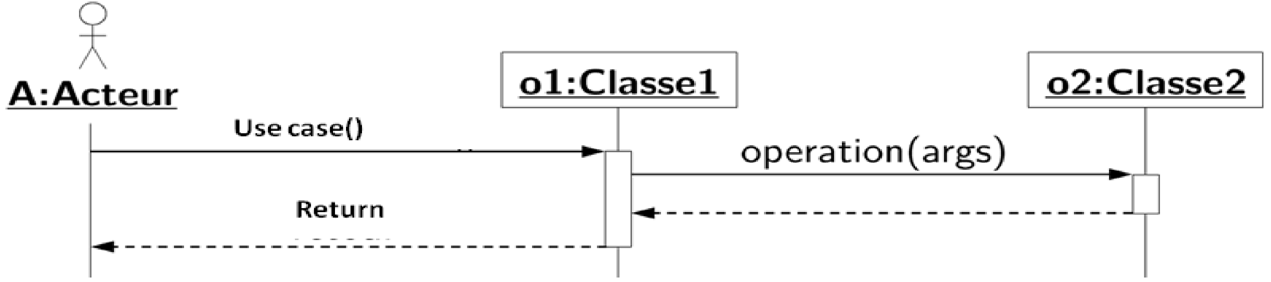


Figure 69 Sequence Diagram

<u>**Lifelines and Message Exchanges in Sequence Diagrams**</u>**:** In a **sequence diagram**, the **lifeline** of each entity (actor or object) is represented as a **vertical dashed line**. This line

illustrates the **existence of the entity over time**, starting from the moment it is created or starts interacting in the scenario.

The **horizontal arrows** between lifelines represent **message exchanges**. These messages can be: **Synchronous** (a response is expected) (**Figure 70**), **Asynchronous** (no immediate response) (**Figure 71**), or **return messages** (dashed lines indicating responses or results).

Vertical axis: Represents **time**, flowing **top to bottom** (the further down, the later the event).

Horizontal axis: Represents **communication** between different **lifelines** through **message passing**.

Figure 70 Synchronous message
Source: BookMyEssay. Synchronous Message in Sequence Diagram. Accessed June 3, 2025

Figure 71 Asynchronous message
Source: BookMyEssay. Asynchronous Message in Sequence Diagram. Accessed June 3, 2025

Objective during the Design Phase: To describe **how a use case is implemented** within the system, based on the structure defined in the **class diagram (Figure 72).**

The goal is to **map the dynamic behavior** (as described in the use case) onto the **static architecture** of the system, identifying which classes are responsible for which parts of the functionality.

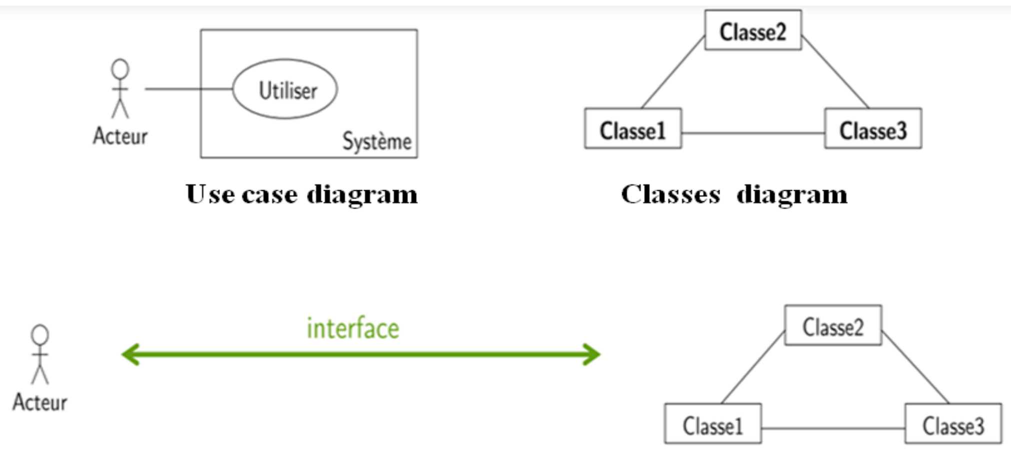


Figure 72 Role of sequence diagram

Object Creation and Destruction in Sequence Diagrams: In UML **sequence diagrams**, it is possible to represent the **creation and destruction of objects** as part of the system's dynamic behavior. An **object is created** when a specific message is sent to instantiate it. This is visually represented by a **dashed lifeline** that begins below the top of the diagram, indicating that the object did not exist before the creation message. The **message that creates the object** is typically a constructor or initialization method, drawn as a solid horizontal arrow pointing to the **object's lifeline at its start**.

On the other hand, the **destruction of an object** is represented by placing a large **"X"** at the end of the object's lifeline. This indicates the moment when the object ceases to exist, usually after receiving a message that triggers its termination or deletion. After this point, the object cannot participate in any further interactions (**Figure 73).**

Representing creation and destruction is particularly useful for understanding **object lifecycles**, **resource management**, and the **scope of interactions** in a system scenario. It helps designers visualize **when objects are active**, and **how long they live** during the execution of a use case.

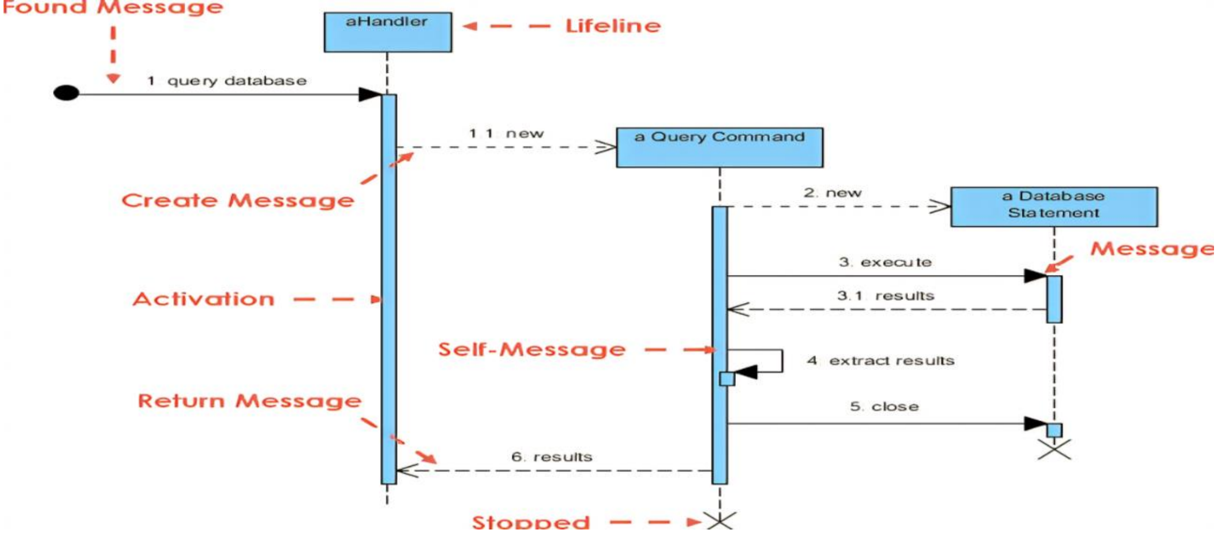


Figure 73 Creation and destruction of an object

Alternatives in Sequence Diagrams: In UML sequence diagrams, the **alternative (alt)** construct is used to model **conditional behavior**, meaning that **only one of several possible message flows** will occur, depending on a specified condition. This construct is useful for representing **"if-else" logic** within the interaction between objects and actors.

The **alt block** is represented as a **frame** (a rectangular container) labeled alt, which is divided into **two or more compartments**, each corresponding to a possible **execution path**. Each compartment is annotated with a **guard condition**, written between square brackets (e.g., [condition1] or [else]), which determines whether the messages inside that compartment will be executed **(Figure 74).**

Only the messages in the **first compartment whose guard evaluates to true** will be executed during a scenario. If none of the conditions hold, and there is no else branch, then no action is taken.

The use of alternatives allows designers to **capture decision logic explicitly**, making system behavior **clearer and more traceable** in scenarios where **multiple possible outcomes** exist.

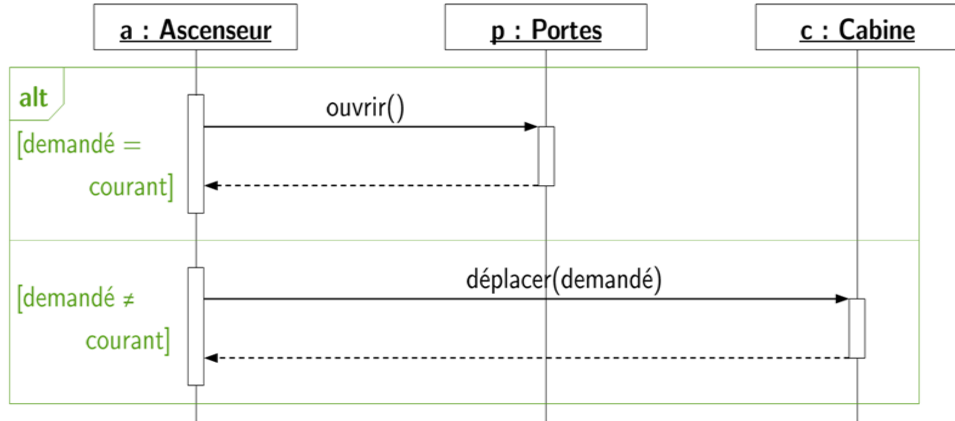


Figure 74 Alternative bloc in a sequence diagram

Loops in Sequence Diagrams: In UML sequence diagrams, a **loop** is used to represent the **repetition of a sequence of messages** under a specified condition. This is particularly useful for modeling **iterative behaviors**, such as traversing a list, retrying an operation, or repeating a request until a certain condition is met (**Figure 75).**

Loops are depicted using a **frame** labeled loop, which surrounds the sequence of messages to be repeated. A **guard condition** is placed inside the loop frame, written in square brackets (e.g., [i < 5] or [while moreItems]), and it determines how long the sequence should be repeated. The loop executes **as long as the condition evaluates to true**.

In some cases, a **note** may be added outside the frame to provide further explanation, such as the number of iterations or a reference to a specific algorithm. Loops in sequence diagrams help clarify **repetitive processes** and ensure that the logic of repeated interactions is explicitly documented in the design.

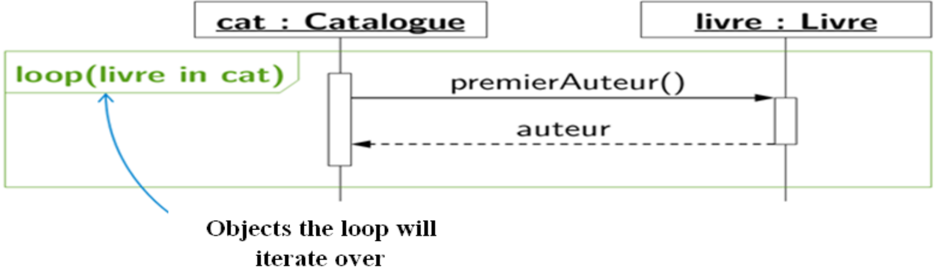


Figure 75 Loop in a sequence diagram

<u>Referencing another Diagram in a Sequence Diagram</u>: In UML sequence diagrams, it is possible to **refer to another interaction or diagram** using a construct called an **interaction use**. This feature allows designers to **reuse predefined interactions** in different contexts, which helps simplify complex diagrams and avoid redundancy.

The reference is represented using a **"ref" frame**, labeled ref, which contains the **name of the referenced interaction** along with any **parameters or conditions** required. This frame acts as a **placeholder** for the detailed sequence of messages described elsewhere (**Figure 76).**

Using interaction references promotes **modularity** and **clarity** by breaking down a large scenario into smaller, more manageable components. It also supports **consistency** across multiple diagrams and makes maintenance easier when updating system behavior.

This technique is particularly useful in large systems where several use cases share common sub-interactions, such as authentication steps, logging, or data validation.

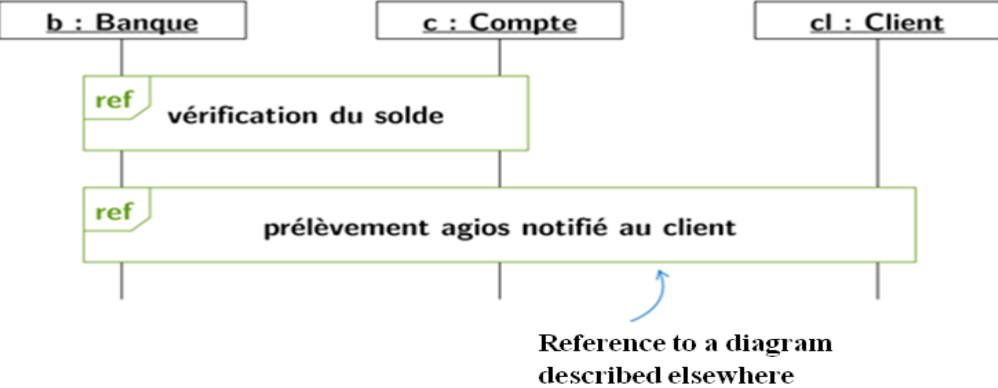


Figure 76 Referencing the Diagram

Exercises

Exercise 1

In a store, a shopkeeper uses a stock management system for articles, with the following functionalities:

-Editing a supplier’s profile.

-Possibility to add a new article, which requires first editing the supplier's profile. If the supplier does not exist, it can then be created.

-Editing the inventory. From this screen, the user can choose to print the inventory, delete an article, or edit an article's profile.

Determine the appropriate use case diagram.

Exercise 2

The fourth-year Industrial Engineering students at ESSAT have decided to create a new club within the school. To organize properly, they established a clear hierarchy, with specific positions assigned tasks and privileges:

- The club president is elected by the club's general assembly from among the candidates running for the position.
- The general assembly is composed of the club's founding members.
- When a member leaves the general assembly, their replacement is elected by the assembly from the candidates running for the position, with conditional approval from the president.
- The club treasurer is appointed by the president with conditional approval from the general assembly and is responsible for managing the club's treasury.
- The general assembly has the sole right to monitor and review the treasurer's work by approving or rejecting the annual financial report presented by the treasurer. The financial report is signed by the president once approved by the assembly.
- The event manager is appointed by the president. They are responsible for booking venues for club events, sending invitations, coordinating with caterers for meals (buffets, table service, coffee breaks, etc.). However, no expenses (e.g., signing a purchase order) can be made without the conditional approval of the treasurer.
- The communications officer manages the club's various communication campaigns, finds partners, attracts sponsors, gathers feedback from different audiences, conducts surveys, etc. However, contract signing with partners or sponsors requires approval from the general assembly.
- The meeting secretary is solely responsible for drafting the minutes of each general assembly meeting. This is a rotating position based on seniority among the assembly members.

The following figure illustrates a UML use case diagram that represents the club's operations. It contains 7 errors. Identify these errors and correct them, explaining why each correction is necessary. (Figure 77)

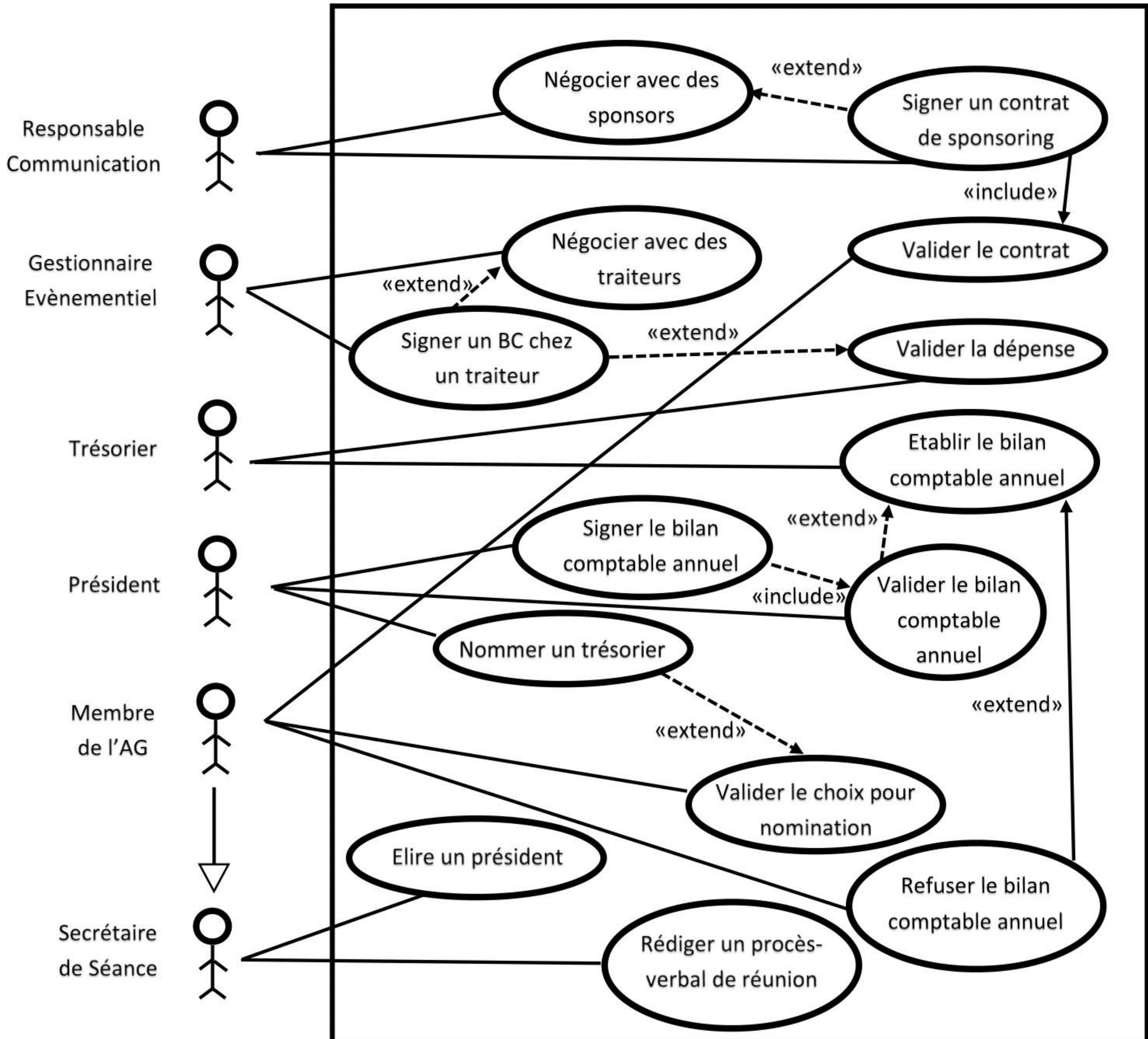


Figure 77 Use case diagram

Exercise 3: Production Workshop Management

Context:

A company manufactures mechanical parts. The information system allows users to:

-plan production orders

-record completed production

-report machine breakdowns

Tasks:

-Identify the system actors

-Identify the main use cases

-Draw the use case diagram

-Add at least one **« include »** or **« extend »** relationship

Exercise 4: Inventory Management System

Context:

An industrial warehouse manages:

-goods reception

-storage

-order preparation

-shipping

Possible actors: Warehouse operator, Logistics manager, Supplier

Tasks:

List all use cases

Identify the interactions between actors and the system

Introduce the following relationships:

« include » (e.g., check stock)

« extend » (e.g., handle stock shortage)

Draw the complete UML diagram

Exercise 5: Industrial Maintenance System

Context:

-A system is used to manage machine maintenance, including:

-preventive maintenance planning

-corrective intervention

-spare parts management

Tasks:

-Identify the actors (technician, maintenance manager, automatic system, etc.)

-Define the use cases

-Add the following relationships: actor generalization (if applicable), include / extend

Propose an improved version of the diagram

Exercise 6: Quality Control System

Context:

In a factory, the quality system is used to:

-perform product inspections

-record defects

-launch corrective actions

-generate quality reports

Tasks:

-Identify all actors (operator, quality manager, customer, etc.)

-Organize the use cases into packages

-Add: include / extend relationships, generalization between use cases

-Discuss which use cases are critical for the company

Exercise 7: Integrated Industrial ERP System

Context:

An industrial ERP system manages:

-production

-logistics

-maintenance

-purchasing

Tasks:

-Decompose the system into subsystems

-Identify internal and external actors

-Build a global use case diagram

Propose:

-a hierarchy of use cases

-complex relationships

-Justify your modeling choices

Bonus Reflection

For each exercise, answer the following questions:

What is the system boundary?

Which use cases are mandatory and which are optional?

What are the risks if the diagram is poorly designed?

Exercise 8:

A company wants to improve its ERP by introducing new functionalities. This concerns two departments:

Human Resources: The company wants to manage its databases of absences, employees, and job descriptions. The company has:A personnel file containing key employee data: Employee ID, Last Name, First Name, Date of Birth, Hiring Date, Job Position.

A job description files containing: Job Code, Job Title, Base Salary, Main Tasks. Upon arrival, employees clock in using a time clock before starting their shift, which automatically records each punch-in in the database. To achieve this, developers want to modify the ERP to display the following interfaces:

Procurement: The company wants to manage its databases of purchase orders, products, and suppliers. The company has: A products file containing key product data: Product Code, Product Type, Weighted Average Price, Stock Quantity.

A suppliers file containing: Supplier Name, Business Name, Address.

Please create UML class diagrams for each of the two modules (Human Resources and Procurement), indicating class names, their attributes, the relationships between them, and their multiplicities.

Note: The two modules (and thus their diagrams) are completely independent from each other.

Exercise 9: Production Order Management

Context: A production system manages manufacturing orders.

A manufacturing order includes the following information:

Reference, date, quantity to produce, status (planned, in progress, completed)

A manufacturing order: concerns one product, uses several raw materials

Tasks:

-Identify the following classes: ManufacturingOrder, Product, RawMaterial

-Add an associative class if necessary

-Define: relationships (1..*, etc.), attributes

-Add methods (e.g., startProduction())

Odoo link:→ mrp.production, mrp.bom

Exercise 10: Inventory Management

Context: The inventory system allows users to:

-track stock inflows and outflows

-manage storage locations

-know available quantities

Tasks:

-Identify the following classes: Product, StockMovement, Location

-Add: movement date, quantity

-Define the relationships: one product → several movements

one movement → one source location and one destination location

-Add the following constraint: quantity ≥ 0

Odoo link:→ stock.move, stock.location

Exercise 11: Industrial Maintenance

Context: Equipment and maintenance management system.

An equipment item includes: name, commissioning date, status

An intervention includes: date, type (preventive, corrective), technician

Tasks:

-Identify the following classes: Equipment, Intervention, Technician

-Add the relationship: one equipment item → several interventions

-Add inheritance: PreventiveIntervention, CorrectiveIntervention

-Add methods: schedule(), close()

Odoo link:→ maintenance.equipment, maintenance.request

Exercise 12: Purchasing Management

Context: A system manages supplier purchase orders.

A purchase order includes: number, date, supplier

An order line includes: product, quantity, price

Tasks: Identify the following classes: PurchaseOrder, OrderLine, Supplier , Product

-Define: a composition relationship between the purchase order and its order lines

-Add the method: calculateTotal()

-Add the following constraint: quantity > 0

Odoo link:→ purchase.order, purchase.order.line

Exercise 13: Mini Industrial ERP System

Context:
You are required to model a mini system integrating: production, inventory, maintenance, purchasing

Tasks:

-Identify all the main classes

-Define: relationships between modules

-Add: inheritance, associative classes

-Propose a modular architecture similar to Odoo

Note: For each exercise, reflect on the following questions:

-Which Odoo model corresponds to each class?

-Which ORM relationship should be used: Many2one, One2many, Many2many

-Which Python fields should be created?

Exercise 14:

Draw the sequence diagram for the use case **"Perform a Personal Transfer"**, based on the class diagram provided below (**Figure 78)**. Then, complete the class diagram if necessary after creating the sequence diagram.

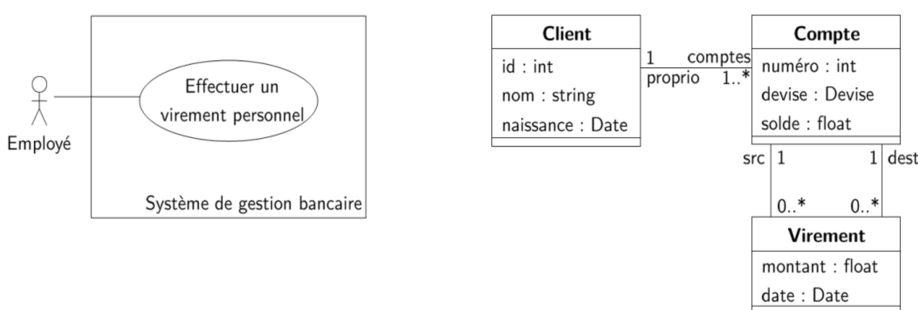


Figure 78 Use case and class diagrams

Mini Project: UML Modeling and Odoo Module Development "School Library Management at ESSAT"

Learning Objectives

The purpose of this mini-project is to enable students to:

-Understand and apply UML modeling concepts

-Design a complete industrial/organizational information system

-Establish the link between UML modeling and concrete development in Odoo ERP

-Develop a functional Odoo module

-Structure an IT project from analysis to implementation

Project Context

ESSAT wishes to digitalize the management of its library in order to improve:

Book tracking, Loan management, Student traceability, The overall organization of the documentary system

At present, library management is semi-manual (Excel files or paper records), which leads to:

Frequent errors, Loss of time, Poor visibility of stock availability

Your mission is to design and develop an Odoo module that allows efficient management of the library

Part 1: UML Analysis and Modeling

Requirements Identification

You must identify the main functionalities of the system.

Examples (non-exhaustive): Book management, Author management, Student management, Book borrowing, Book return, Late return management, Book search

Task:

-Write a simplified requirements specification document

Use Case Diagram

Identify the actors and their interactions with the system.

Possible actors: Student, Librarian, Administrator

Task:

-Create a UML use case diagram

-Identify **include** / **extend** relationships

Class Diagram

Build a complete data model.

Possible classes: Book, Author, Loan, Student, Category

Task:

- Define attributes and methods
- Specify relationships and cardinalities
- Add inheritance if relevant

Sequence Diagram

Model a key scenario.

Example: Borrowing a book

Task:

-Identify the objects involved

-Describe the interactions over time

Activity Diagram

Describe the process flow.

Example: Borrowing and return process

Task:

-Define the different steps

-Add decisions and conditions

State Diagram (Optional but Recommended)

Example: States of a book (*available, borrowed, reserved*)

Part 2: Odoo-Oriented Design

You must establish the connection between UML and Odoo.

Task:

For each UML class:

-Identify the corresponding Odoo model

-Define the fields, such as: Char, Many2one, One2many, Many2many

Example: Book → model library.book, Author → model library.author

Part 3: Odoo Module Development

1. Module Creation

Create a module named: essat_library

Define the standard Odoo module structure

2. Models (models.py)

Implement the following models: Book, Author, Student, Loan

Include: ORM relationships, Constraints (e.g., book availability)

3. Views (XML)

Create: Tree views, Form views, Menus and actions

4. Business Logic

Implement the following functionalities: Borrow a book, Return a book, Check availability

5. Security

-Define user groups

-Add access control rules

Part 4: Testing and Validation

Task:

-Test the main scenarios

-Check possible errors

-Propose improvements

Expected Deliverables

Students are expected to provide:

1. **A report (PDF)** containing: Requirements analysis, All UML diagrams, Explanation of modeling and design choices
2. **The Odoo module source code**
3. **An oral presentation**

Constraints and Recommendations

-Follow UML standards

-Use clear and meaningful names for classes and relationships

-Adopt a clean architecture

-Test regularly throughout the project

-Bonus (Optional): Add a reservation system, Add late return notifications, Add a dashboard

7. Conclusion

UML modeling is an essential step in the design of information systems, particularly in industrial environments where process complexity requires a clear, structured representation shared among all project stakeholders.

In this chapter, we introduced the fundamental principles of the UML language, as well as three key diagrams: use case diagrams, class diagrams, and sequence diagrams. Each of these diagrams provides a complementary perspective on the system: functional, structural, and dynamic. Together, they offer a comprehensive and coherent view of the system to be designed.

The use of UML is not limited to documentation; it plays a central role in requirements analysis, communication among stakeholders, and preparation for technical implementation. In the context of ERP development, particularly with Odoo, UML diagrams facilitate the transition from abstract design to the concrete implementation of modules by clearly identifying models, relationships, and interaction flows.

The exercises presented in this chapter have enabled the practical application of these concepts in realistic industrial contexts, thereby strengthening students' ability to model complex systems and structure their thinking.

In conclusion, mastering UML is a fundamental skill for any information systems engineer. It not only enables the design of robust and scalable solutions but also ensures better understanding and collaboration throughout the project lifecycle, from analysis to development and maintenance.

References:

[1]: Logistiikan Maailma. (2024, December 30). *Information, money and material flow*. Retrieved from https://www.logistiikanmaailma.fi/en/logistics/logistics-and-supply-chain/information-money-and-material-flow/

[2]: Laudon, K. C., & Laudon, J. P. (2020). *Management Information Systems: Managing the Digital Firm* (16th Edition). Pearson.

[3]: SOLIDitech. (2021, July 13). *An Introduction to Centralised Databases*. The SOLID Blog. https://blog.soliditech.com/blog/an-introduction-to-a-centralised-databases

[4]: Monk, E., & Wagner, B. (2012). *Concepts in Enterprise Resource Planning* (4th Edition). Cengage Learning.

[5]: Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley.

[6]: Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

[7]: **BookMyEssay.** (2021, January). Synchronous message in sequence diagram [Image]. BookMyEssay. https://www.bookmyessay.co.uk/wp-content/uploads/2021/01/Synchronous-Message-768x513.jpg

[8]: BookMyEssay. (2021, January). [Asynchronous message in sequence diagram] [Image]. BookMyEssay. [https://www.bookmyessay.co.uk/blog/visualizing-system-design-using-uml-based-sequence-diagrams/]