Contents

[1 Introduction 2](#_Toc110714594)

[1.1 Information Systems 2](#_Toc110714595)

[1.1.1 Information System Definition 2](#_Toc110714596)

[1.1.2 Components of Information Systems 3](#_Toc110714597)

[1.1.3 The Systems Development Life Cycle 5](#_Toc110714598)

[1.1.4 Types of software and their differences 9](#_Toc110714599)

[1.2 Software engineering 9](#_Toc110714600)

[1.2.1 Stakeholders in software engineering 10](#_Toc110714601)

[1.2.2 Attributes of software quality 11](#_Toc110714602)

[1.2.3 Activities common to software projects 12](#_Toc110714603)

[1.2.4 Design 13](#_Toc110714604)

[1.2.5 Modeling 13](#_Toc110714605)

[1.2.6 Programming 14](#_Toc110714606)

[1.2.7 Quality assurance 14](#_Toc110714607)

[1.2.8 Deployment 14](#_Toc110714608)

[2 Laboratory work #1. Conceptual modeling using UML 15](#_Toc110714609)

[2.1 UML Modeling 15](#_Toc110714610)

[2.2 UML class diagrams 16](#_Toc110714611)

[2.2.1 Attributes characteristics 17](#_Toc110714612)

[2.2.2 Relationships 17](#_Toc110714613)

[2.3 Class Diagram in Software Development Lifecycle 20](#_Toc110714614)

[2.4 Best practices of Designing of the Class Diagram 20](#_Toc110714615)

[2.5 Using diagrams.net to draw UML Class Diagram 21](#_Toc110714616)

[2.5.1 Work with class shapes 23](#_Toc110714617)

[2.5.2 Editing UML classes in diagrams.net 24](#_Toc110714618)

[2.5.3 Add an attribute or method 24](#_Toc110714619)

[2.5.4 Attribute and method names in UML notation 25](#_Toc110714620)

[2.5.5 Connectors for relationships in UML class diagrams 26](#_Toc110714621)

[2.6 The assignment 28](#_Toc110714622)

[3 Laboratory work #2. Database modeling and design 29](#_Toc110714623)

[3.1 Entity-Relational Diagrams (ERD) 29](#_Toc110714624)

[3.1.1 Example of Entity-Relationship Diagram 33](#_Toc110714625)

[3.1.2 The algorithm of translation ERD to relational database schema 34](#_Toc110714626)

[3.2 Database design using pgAdmin 4 35](#_Toc110714627)

[3.2.1 Create Table Example 37](#_Toc110714628)

[3.2.2 Query Tool Example 39](#_Toc110714629)

[3.2.3 pgAdmin 4 ERD Tool 41](#_Toc110714630)

# Introduction

## Information Systems

Combining hardware, software, human power and processes, an ***information system*** refers to a network used to collect, store, process, analyze and distribute data. Information systems and professionals with advanced degrees in information systems can help businesses and other organizations improve their efficiency, maximize revenue and streamline their operations.

### Information System Definition

Many people think of information systems as computer-based technology. While information systems often incorporate computers to help manage data and achieve business objectives, they do not necessarily have to include computers.

There are different types of information systems that can serve a variety of purposes depending on an organization’s needs. Examples include:

**Data warehouses**. Data warehouses are data management systems that support analytics and other business intelligence activities. They consolidate and analyze data from a large variety of sources. Data warehouses can provide insight into businesses to help improve decision-making.

**Enterprise systems**. Enterprise systems, also known as enterprise resource planning (ERP) systems, are integrated systems that combine all the hardware and software a business uses for different functions in its operations. These organization-wide systems help information flow between departments and allow processes from different parts of the business to be integrated across a company.

**Expert systems**. Expert systems use artificial intelligence to mimic human decision-making. The software uses human knowledge to solve problems that would typically require a person’s expertise. Expert systems can be applied in areas such as medical diagnoses, accounting and coding.

**Geographic information systems**. Geographic information systems (GIS) are tools that gather, organize, map and analyze data with a spatial component. GIS can improve analysis and decision-making by allowing users to visualize data on a map. Global information systems are a type of GIS that synthesize worldwide data.

**Office automation systems**. Office automation systems combine communication technology, people and computers to help perform office activities, such as preparing written communication, printing, scheduling or creating reports.

### Components of Information Systems

Every information system includes several key components: hardware, software, telecommunications, people and data. Hardware refers to the physical pieces of the information system; software is the programming that controls the information system; telecommunication transmits information through the system; humans manage and interact with the information system; and data is information stored within and processed by the system.

**Hardware**

The hardware component of an information system comprises the physical elements of the system. People can touch and feel pieces of hardware. These mechanisms, equipment and wiring allow systems like computers, smartphones and tablets to function.

Input and output devices are essential pieces of technology that allow humans to interact with computers and other information systems. Keyboards, mice, microphones and scanners are all examples of input devices. And output devices might include printers, monitors, speakers and sound and video cards.

Pieces of hardware including microprocessors, hard drives, electric power supply units, and removable storage also allow computers to store and process data.

**Software**

Software are the intangible programs that manage information system functions, including input, output, processing and storage.

System software – such as the MacOS or Microsoft Windows operating systems – provides a base for application software to run.

Application software operates programs geared toward particular uses in information systems. For example, word processing applications are used to create and edit text documents. Graphical user interface (GUI) software is among the most common application software; it presents the information stored in computers and allows users to interact with computers through digital graphics – such as icons, buttons and scroll bars – rather than through text-based commands.

Software can be either open source or closed source. Open source software coding is publicly available for users and programmers to manipulate, whereas closed source software is proprietary.

**Telecommunications**

Telecommunications systems connect computer networks and allow information to be transmitted through them. Telecommunications networks also allow computers and storage services to access information from the cloud.

There are a number of methods telecommunications networks use to convey information. Coaxial cables and fiber optic cables are used by telephone, internet and cable providers to transmit data, video and audio messages.

Local-area networks (LANs) connect computers to create computer networks in a designated space, like a school or home. Wide-area networks (WANs) are collections of LANs that facilitate data-sharing across large areas. A virtual private network (VPN) allows a user to protect their online privacy by encrypting data on public networks.

Microwaves and radio waves can also be used to transmit information in telecommunications networks.

**Data**

Data are intangible, raw facts that are stored, transmitted, analyzed and processed by other components of information systems. Data are often stored as numerical facts, and they represent quantitative or qualitative information.

Data can be stored in a database or data warehouse, in a form that best suits the organization using it.

Databases house collections of data that can be queried or retrieved for specific purposes. Databases allow users to perform fundamental operations, such as storage and retrieval. Data warehouses, on the other hand, store data from multiple sources for analytical purposes. They allow users to assess an organization or its operations.

**Human Resources**

Human resources are a crucial part of information systems. The human component of information systems encompasses the qualified people who influence and manipulate the data, software and processes in information systems. Humans involved in information systems may include business analysts, information security analysts or system analysts.

*Business analysts* work to elevate an organization’s operations and processes. They often focus on improving efficiency and productivity or streamlining distribution. *Information security analysts* work to prevent data breaches and cybersecurity attacks. And *system analysts* use information technology to help organizations optimize their user experiences with programs.

### The Systems Development Life Cycle

In many ways, building an information system is similar to building a house. First, the house (or the information system) starts with a basic idea. Second, this idea is transformed into a simple drawing that is shown to the customer and refined (often through several drawings, each improving on the last) until the customer agrees that the picture depicts what he or she wants. Third, a set of blueprints is designed that presents much more detailed information about the house (e.g., the type of water faucets or where the telephone jacks will be placed). Finally, the house is built following the blueprints, often with some changes directed by the customer as the house is erected. The SDLC has a similar set of four fundamental phases: *planning*, *analysis*, *design*, and *implementation*. Different projects might emphasize different parts of the SDLC or approach the SDLC phases in different ways, but all projects have elements of these four phases. Each phase is itself composed of a series of steps, which rely upon techniques that produce deliverables (specific documents and files that provide understanding about the project).

In many projects, the SDLC phases and steps proceed in a logical path from start to finish. In other projects, the project teams move through the steps consecutively, incrementally, iteratively, or in other patterns. There are two important points to understand about the SDLC. First, you should get a general sense of the phases and steps through which IS projects move and some of the techniques that produce certain deliverables. Second, it is important to understand that the SDLC is a process of gradual refinement. The deliverables produced in the analysis phase provide a general idea of the shape of the new system. These deliverables are used as input to the design phase, which then refines them to produce a set of deliverables that describes in much more detailed terms exactly how the system will be built. These deliverables, in turn, are used in the implementation phase to produce the actual system. Each phase refines and elaborates on the work done previously.

**Planning**

The planning phase is the fundamental process of understanding why an information system should be built and determining how the project team will go about building it. It has two steps:

1. During project initiation, the system’s business value to the organization is identified: How will it lower costs or increase revenues? Most ideas for new systems come from outside the IS area (e.g., from the marketing department, accounting department) in the form of a system request. A system request presents a brief summary of a business need, and it explains how a system that supports the need will create business value. The IS department works together with the person or department that generated the request (called the project sponsor) to conduct a feasibility analysis. The system request and feasibility analysis are presented to an information systems approval committee (sometimes called a steering committee), which decides whether the project should be undertaken.

2. Once the project is approved, it enters project management. During project management, the project manager creates a workplan, staffs the project, and puts techniques in place to help the project team control and direct the project through the entire SDLC. The deliverable for project management is a project plan, which describes how the project team will go about developing the system.

**Analysis**

The analysis phase answers the questions of *who* will use the system, *what* the system will do, and *where* and *when* it will be used. During this phase, the project team investigates any current system(s), identifies opportunities for improvement, and develops a concept for the new system. Th is phase has three steps:

1. An analysis strategy is developed to guide the project team’s efforts. Such a strategy usually includes an analysis of the current system (called the as-is system) and its problems and then ways to design a new system (called the to-be system).

2. The next step is requirements gathering (e.g., through interviews or questionnaires). The analysis of this information—in conjunction with input from the project sponsor and many other people—leads to the development of a concept for a new system. The system concept is then used as a basis to develop a set of business analysis models, which describe how the business will operate if the new system is developed.

3. The analyses, system concept, and models are combined into a document called the system proposal, which is presented to the project sponsor and other key decision makers (e.g., members of the approval committee) who decide whether the project should continue to move forward. The system proposal is the initial deliverable that describes what business requirements the new system should meet. Just keep in mind that the deliverable from the analysis phase is both an analysis and a high-level initial design for the new system.

**Design**

The design phase decides how the system will operate, in terms of the hardware, software, and network infrastructure; the user interface, forms, and reports; and the specific programs, databases, and files that will be needed. Although most of the strategic decisions about the system were made in the development of the system concept during the analysis phase, the steps in the design phase determine exactly how the system will operate. The design phase has four steps:

1. The design strategy is first developed. It clarifies whether the system will be developed by the company’s own programmers, whether the system will be outsourced to another firm (usually a consulting firm), or whether the company will buy an existing software package.

2. This leads to the development of the basic architecture design for the system, which describes the hardware, software, and network infrastructure to be used. In most cases, the system will add or change the infrastructure that already exists in the organization. The interface design specifies how the users will move through the system (e.g., navigation methods such as menus and on-screen buttons) and the forms and reports that the system will use.

3. The database and file specifications are developed. These define exactly what data will be stored and where they will be stored.

4. The analyst team develops the program design, which defines the programs that need to be written and exactly what each program will do. This collection of deliverables (architecture design, interface design, database and file specifications, and program design) is the system specification that is handed to the programming team for implementation. At the end of the design phase, the feasibility analysis and project plan are reexamined and revised, and another decision is made by the project sponsor and approval committee about whether to terminate the project or continue.

**Implementation**

The final phase in the SDLC is the implementation phase, during which the system is actually built (or purchased, in the case of a packaged software design). This is the phase that usually Systems Development Methodologies. This phase has three steps:

1. System construction is the first step. The system is built and tested to ensure that it performs as designed. Because the cost of bugs can be immense, testing is one of the most critical steps in implementation. Most organizations give more time and attention to testing than to writing the programs in the first place.

2. The system is installed. Installation is the process by which the old system is turned off and the new one is turned on. One of the most important aspects of conversion is the development of a training plan to teach users how to use the new system and help manage the changes caused by the new system.

3. The analyst team establishes a support plan for the system. This plan usually includes a formal or informal post-implementation review as well as a systematic way for identifying major and minor changes needed for the system.

### Types of software and their differences

There are many different types of software. One of the most important distinctions is between custom software, generic software and embedded software. Custom software is developed to meet the specific needs of a particular customer and tends to be of little use to others (although in some cases developing custom software might reveal a problem shared by several similar organizations). Much custom software is developed in-house within the same organization that uses it; in other cases, the development is contracted out to consulting companies. Custom software is typically used by only a few people and its success depends on meeting their needs. Examples of custom software include web sites, air-traffic control systems and software for managing the specialized finances of large organizations. Generic software, on the other hand, is designed to be sold on the open market, to perform functions that many people need, and to run on general purpose computers. Requirements are determined largely by market research. There is a tendency in the business world to attempt to use generic software instead of custom software because it can be far cheaper and more reliable. The main difficulty is that it might not fully meet the organization’s specific needs. Generic software is often called Commercial Off-The-Shelf software (COTS), and it is sometimes also called shrink-wrapped software since it is commonly sold in packages wrapped in plastic. Generic software producers hope that they will sell many copies, but their success is at the mercy of market forces. Examples of generic software include word processors, spreadsheets, compilers, web browsers, operating systems, computer games and accounting packages for small businesses. Embedded software runs specific hardware devices which are typically sold on the open market. Such devices include washing machines, DVD players, microwave ovens and automobiles. Unlike generic software, users cannot usually replace embedded software or upgrade it without also replacing the hardware. The open-market nature of the hardware devices means that developing embedded software has similarities to developing generic software; however, we place it in a different category due to the distinct processes used to develop it.

## Software engineering

***Software engineering*** is the process of solving customers’ problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

Solving customers’ problems should be the goal of every software engineering project. Before finalizing any software engineering decision, you should therefore ask yourself whether the proposed alternative will help achieve this goal. In particular, it is important to recognize activities that are not consistent with this goal, such as adding unnecessary features. Software engineers have the responsibility to recognize situations when it would be most cost effective not to develop software at all, to develop simpler software or to purchase existing software.

### Stakeholders in software engineering

Many people are involved in a software engineering project and expect to benefit from its success. We will classify these stakeholders into four major categories, or roles, each having different motivations, and seeing the software engineering process somewhat differently.

**Users**. These are the people who will use the software. Their goals usually include doing enjoyable or interesting work, and gaining recognition for the work they have done. Often, they will welcome new or improved software, although some might fear it could jeopardize their jobs. Users appreciate software that is easy to learn and use, makes their life easier, helps them achieve more, or allows them to have fun.

**Customers** (also known as clients). These are the people who make the decisions about ordering and paying for the software. They may or may not be users – the users may work for them. Their goal is either to increase profits or simply to run their business more effectively. Customers appreciate software that helps their organization save or make money, typically by improving the productivity of the users and the organization as a whole. If you are developing custom software, then you know who your customers are; if you are developing generic software, then you often only have potential customers in mind.

**Software developers**. These are the people who develop and maintain the software, many of whom may be called software engineers. Within the development team there are often specialized roles, including requirements specialists, database specialists, technical writers, configuration management specialists, etc. Development team members normally desire rewarding careers, although some are more motivated by the challenge of solving difficult problems or by being a well-respected ‘guru’ in a certain area of expertise. Many developers are motivated by the recognition they receive by doing high quality work.

**Development managers**. These are the people who run the organization that is developing the software; they often have an educational background in business administration. Their goal is to please the customer or sell the most software, while spending the least money. It is important that they have considerable knowledge about how to manage software projects, but they may not be as intimately familiar with small details of the project as are some of the software developers. For this reason, it is important that software developers keep their managers informed of any problems.

### Attributes of software quality

The following are five of the most important attributes of software quality. Software engineers try to balance the relative importance of these attributes so as to design systems with the best overall quality, as limited by the money and time available.

**Usability**. The higher the usability of software, the easier it is for users to work with it. There are several aspects of usability, including learnability for novices, efficiency of use for experts, and handling of errors.

**Efficiency**. The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth and other resources. This is important to customers in order to reduce their costs of running the software, although with today’s powerful computers, CPU-time, memory and disk usage are less of a concern than in years gone by.

**Reliability**. Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily.

**Maintainability**. This is the ease with which you can change the software. The more difficult it is to make a change, the lower the maintainability. Software engineers can design highly maintainable software by anticipating future changes and adding flexibility. Software that is more maintainable can result in reduced costs for both developers and customers.

**Reusability**. A software component is reusable if it can be used in several different systems with little or no modification. High reusability can reduce the long-term costs faced by the development team. All of these attributes of quality are important. However, the relative importance of each will vary from stakeholder to stakeholder and from system to system. For example, reliability and efficiency are usually both of concern to customers and users; however, in a safety-critical system for controlling a nuclear power plant, reliability would be far more important than efficiency – assuming that faster hardware could be bought if efficiency became a problem.

### Activities common to software projects

The following subsections briefly describe many of the activities commonly found in software engineering projects.

#### Requirements and specification

In order to solve the customer’s problems, you must first understand the problems, the customer’s business environment, and the available technology which can be used to solve the problems. Once you have done this, you can meet with the customers and users to decide on a course of action that will solve the problems. If you decide that developing or modifying software is the best course of action, then you can decide in detail what facilities the software should provide. This overall process may include the following activities.

**Domain analysis**: understanding the background needed so as to be able to understand the problem and make intelligent decisions.

**Defining the problem**: narrowing down the scope of the system by determining the precise problem that needs solving.

**Requirements gathering**: obtaining all the ideas people have about what the software should do.

**Requirements analysis**: organizing the information that has been gathered, and making decisions about what in fact the software should do. The term ‘requirements analysis’ is often used more broadly to include some of the other steps in this list.

**Requirements specification**: writing a precise set of instructions that define what the software should do. These instructions should describe how the software behaves from the perspective of the user, but should not describe any details of the implementation. One of the most important principles of requirements is to separate the ‘what’ from the ‘how’. The ‘what’ refers to the requirements – what is needed to solve the problem. The ‘how’ refers to how the solution will be designed and implemented.

### Design

Design is the process of deciding how the requirements should be implemented using the available technology. Important activities during design include:

■ Deciding what requirements should be implemented in hardware and what in software. This is called systems engineering and is normally only necessary for embedded and other real-time systems. Even for these systems, there is a trend towards implementing more and more facilities in software so that the hardware can be simpler and more generic.

* Deciding how the software is to be divided into subsystems and how the subsystems are to interact. This process is often called software architecture; there are several well-known ways of structuring software which are called architectural patterns or styles.
* Deciding how to construct the details of each subsystem. Such details include the data structures, classes, algorithms and procedures. This process is often called detailed design.
* Deciding in detail how the user is to interact with the system, and the look and feel of the system.
* Deciding how the data will be stored on disk in databases or files.

### Modeling

Modeling is the process of creating a representation of the domain or the software. Various modeling approaches can be used during both requirements analysis and design. These include:

**Use case modeling**. This involves representing the sequences of actions performed by the users of the software.

**Structural modeling**. This involves representing such things as the classes and objects present in the domain or in the software.

**Dynamic and behavioral modeling**. This involves representing such things as the states that the system can be in, the activities it can perform, and how its components interact. Modeling can be performed *visually*, using diagrams, or else using *semi-formal* or *formal* languages that express the information systematically or mathematically. In this guideline, we will primarily use semi-formal notations and diagrams – in particular a visual language called UML.

### Programming

Programming is an integral part of software engineering. It involves the translation of higher-level designs into particular programming languages. It should be thought of as the final stage of design because it involves making decisions about the appropriate use of programming language constructs, variable declarations etc. Most people who call themselves programmers also perform many higher-level design activities. People who limit their work to programming (i.e. who do no higher-level design or analysis) are often today called ‘coders’. One of the objectives of software engineering researchers has been to automate programming. There has been some success in this regard – some tools now generate much of the code for you from models typically represented in UML. However, there will always be a need for some programming done by humans.

### Quality assurance

Quality assurance (QA) encompasses all the processes needed to ensure that the quality objectives are met. Quality assurance occurs throughout a project, and includes many activities, including the following:

* Reviews and inspections. These are formal meetings organized to discuss requirements, designs or code to see if they are satisfactory.
* Testing. This is the process of systematically executing the software to see if it behaves as expected. Quality assurance is also often divided into validation, which is the process of determining whether the requirements will solve the customer’s problem, and verification, which is the process of making sure the requirements have been adhered to.

### Deployment

Deployment involves distributing and installing the software and any other components of the system such as databases, special hardware etc. It also involves managing the transition from any previous system. Deploying a new release of a large system with many users can pose great difficulties – the amount of work is often under-estimated. To keep this book short, we have decided not to discuss deployment.

# Laboratory work #1. Conceptual modeling using UML

## UML Modeling

UML class diagrams are one of the most important tools for both requirements analysis and design of object-oriented software systems. These diagrams show the classes, their attributes and operations as well as the various types of relationships that exist among the classes.

The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. It was developed in the mid-1990s as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson, each of whom had developed their own notation in the early 1990s. The ‘U’ in UML stands for ‘unified’, since its three developers combined the best features of the languages they had each previously developed. The custodian of the UML standard is the Object Management Group (OMG). In 2004 the OMG approved version 2.0 of UML. UML contains a variety of diagram types, including:

**Class diagrams**, which describe classes and their relationships.

**Interaction diagrams**, which show the behavior of systems in terms of how objects interact with each other.

**State diagrams and activity diagrams**, which show how systems behave.

**Component and deployment diagrams**, which show how the various components of systems are arranged logically and physically.

Most systems are therefore documented with the use of diagrams. These provide views of structure and functionality that would be difficult to grasp by looking at code or textual descriptions alone. In other words, diagrams provide abstraction.

A *model* goes beyond a mere set of diagrams. A model captures an interrelated set of information about the system: a diagram is simply one view of that information. Several diagrams can present the same information in slightly different ways, either with different notations or with different levels of detail. I can delete an element from a diagram, and keep it in the model; if I delete an element from the model it should disappear from all diagrams. A model can lead software engineers to have insights about the system; they can analyze the model (manually or using tools) to discover problems and other properties of it. Simple diagrams generated from the model can also help communicate with clients and users. However, it is up to the modeler to generate these easy-to-understand views. Employing UML, a well-defined standard modeling language, adds additional advantages:

* Since it is a standard notation, everybody who looks at the model will be able to interpret it the same way.
* There is a wide variety of tools available to build UML models and to enable simulation, animation and/or generation of code for all or parts of a system.

## UML class diagrams

The main symbols shown on class diagrams are:

**Classes**, which represent the types of data themselves.

**Associations**, which show how instances of classes reference instances of other classes.

**Attributes**, which are simple data found in instances.

**Operations**, which represent the functions performed by the instances.

**Generalizations**, which are used to arrange classes into inheritance hierarchies.

A *class* is represented as a box with the name of the class inside. The name should always be singular and start with a capital letter. When you draw a class in a class diagram, you are saying that the system will contain a class by that name, and that when the system runs, instances of that class will be created. Optionally, the class diagram may also show the attributes and operations contained in each class. This is done by dividing a class box into two or three smaller boxes: the top box contains the class name, the next box lists attributes, and the bottom box lists operations. If you do not want to specify attributes or operations, then you simply omit the box.

Figure 2.1 illustrates how a class can be drawn at several different levels of detail. How much detail you show depends on the phase of development and on what you wish to communicate.

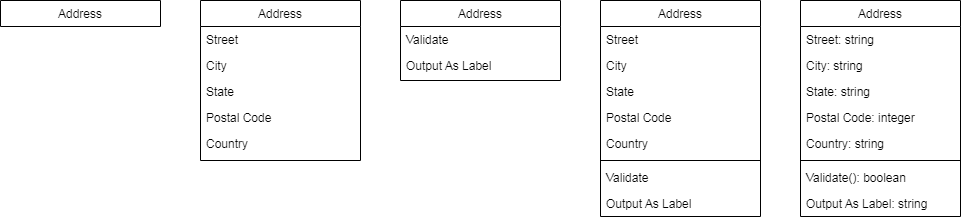


Figure 2.1 The **Address** class at several different levels of detail

### Attributes characteristics

* The attributes are generally written along with the visibility factor.
* Public, private, protected and package are the four visibilities which are denoted by +, -, #, or ~ signs respectively.
* Visibility describes the accessibility of an attribute of a class.
* Attributes must have a meaningful name that describes the use of it in a class.

### Relationships

There are mainly three kinds of relationships in UML:

* Dependencies
* Generalizations
* Associations

#### Dependency

A dependency means the relation between two or more classes in which a change in one may force changes in the other. However, it will always create a weaker relationship. Dependency indicates that one class depends on another (see Figure 2.2).



Figure 2.2 Dependency Relationship

#### Generalization

Generalization relationship is utilized in class, component, deployment, and use case diagrams to specify that the child inherits actions, characteristics, and relationships from its parent.

The generalization relationship is incorporated to record attributes, operations, and relationships in a parent model element so that it can be inherited in one or more child model elements (see Figure 2.3).

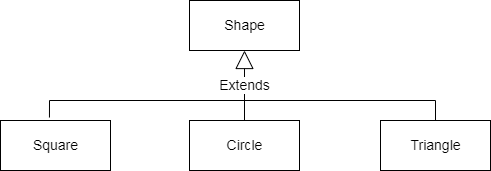


Figure 2.3 Generalization

#### Associations and multiplicity

An association is used to show how instances of two classes will reference each other. The association is drawn as a line between the classes. Symbols indicating multiplicity are shown at each end of the association. The multiplicity indicates how many instances of the class at this end of the association can be linked to an instance of the class at the other end of the association. Figure 1.2 gives some examples of associations, showing their multiplicity. A multiplicity of 1 indicates that there must be exactly one instance linked to each object at the other end of the association. For example, there can only be one Company associated with each Employee in Figure 2.3:

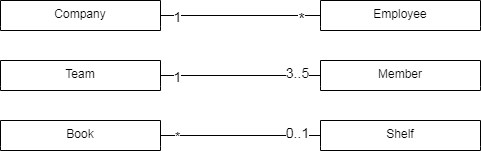


Figure 2.4. Examples of possible multiplicities

A multiplicity of 1 indicates that there must be exactly one instance linked to each object at the other end of the association. A very common multiplicity is ∗, which is normally read as ‘many’, and means any integer greater than or equal to zero. In Figure 1.2, for example, many employees can be associated with a company. If there can be either zero or one object linked to an object at the other end of the association, then the multiplicity is said to be ‘optional’, and the notation 0..1 is used. You can also specify the multiplicity to be an interval, which is shown as two dots between the lower and upper bound. An interval is also sometimes called a range. For example, in a single team there could be from 3 to 5 members (see Figure 2.5).

#### Labeling associations

Each association can be labeled, to make explicit the nature of the association. There are two types of labels, association names and role names. Figure 2.5 shows the same associations as in Figure 2.4, but with labels added.

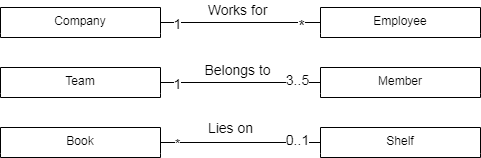


Figure 2.5. Examples of possible multiplicities with labels

An association name should be a verb or verb phrase, and is placed next to the middle of the association. One class becomes the subject and the other class becomes the object of the verb. For example, the association between Employee and Company is called *worksFor*. You can read the association in one direction as, ‘an employee works for a company’.

#### Composition

The composition is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.

As you can see from the example given below, the composition association relationship connects the Person class with Brain class, Heart class, and Arms class. If the person is destroyed, the brain, heart, and legs will also get discarded (see Figure 2.6).

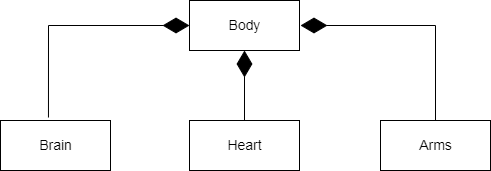


Figure 2.6 Composition of a human body

## Class Diagram in Software Development Lifecycle

Class diagrams can be used in various software development phases. It helps in modeling class diagrams in three different perspectives.

**1. Conceptual perspective:**Conceptual diagrams are describing things in the real world. You should draw a diagram that represents the concepts in the domain under study. These concepts related to class and it is always language-independent.

**2. Specification perspective:** Specification perspective describes software abstractions or components with specifications and interfaces. However, it does not give any commitment to specific implementation.

**3. Implementation perspective:**This type of class diagrams is used for implementations in a specific language or application. Implementation perspective, use for software implementation.

## Best practices of Designing of the Class Diagram

Class diagrams are the most important UML diagrams used for software application development. There are many properties which should be considered while drawing a Class Diagram. They represent various aspects of a software application.

Here, are some points which should be kept in mind while drawing a class diagram:

* The name given to the class diagram must be meaningful. Moreover, It should describe the real aspect of the system.
* The relationship between each element needs to be identified in advance.
* The responsibility for every class needs to be identified.
* For every class, minimum number of properties should be specified. Therefore, unwanted properties can easily make the diagram complicated.
* User notes should be included whenever you need to define some aspect of the diagram. At the end of the drawing, it must be understandable for the software development team.
* Lastly, before creating the final version, the diagram needs to be drawn on plain paper. Moreover, it should be reworked until it is ready for final submission.

## Using diagrams.net to draw UML Class Diagram

There are many tools available in the market to generate UML diagrams. Some are desktop based while others can be used online. Following is a curated list of tools which can be used for the creation of UML models: [Edraw Max](https://bit.ly/39C6v3Q), [Moqups](https://moqups.grsm.io/uml), [MS Visio](https://guru99.live/Z5V3Ur) etc. In this guideline we’ll use the one of the online tools, [diagrams.net](https://www.diagrams.net/) (formerly, draw.io).

In order to draw UML class diagrams in diagrams.net firstly you need to enable the UML shape libraries:

1. Click on More Shapes at the bottom of the left panel.
2. Enable the UML 2.5 and UML shape libraries in the Software section.
3. Click Apply to return to the diagram editor.

Then you’ll see the following screen (see Figure 2.7):

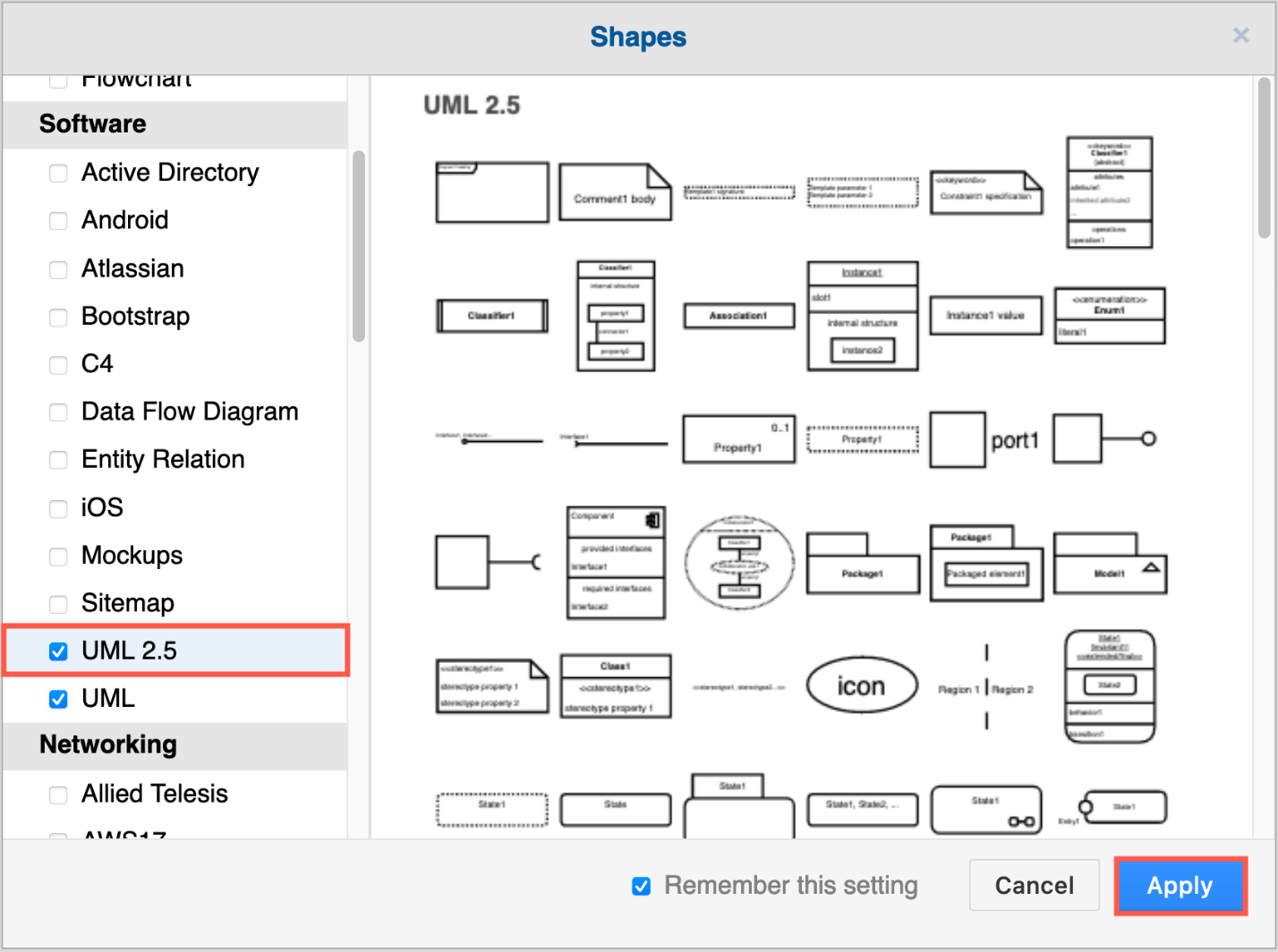


Figure 2.7 Adding UML 2.5 shapes to diagrams.net

Most class shapes are in the main UML shape library. Hover over any shape in the shape library to see a larger preview.

The next step is to use a template or example class diagram:

1. Start a new diagram, or click Arrange > Insert > Template to open the template manager.
2. There is a simple example in the Basic or Default template category.
3. Click on the magnifying glass to see a larger preview.
4. Software category: complex class diagrams.
5. UML category: higher level class diagrams, and package diagrams.
6. Select a template, and click Create or Insert.

You’ll see the following screen (Figure 2.8):

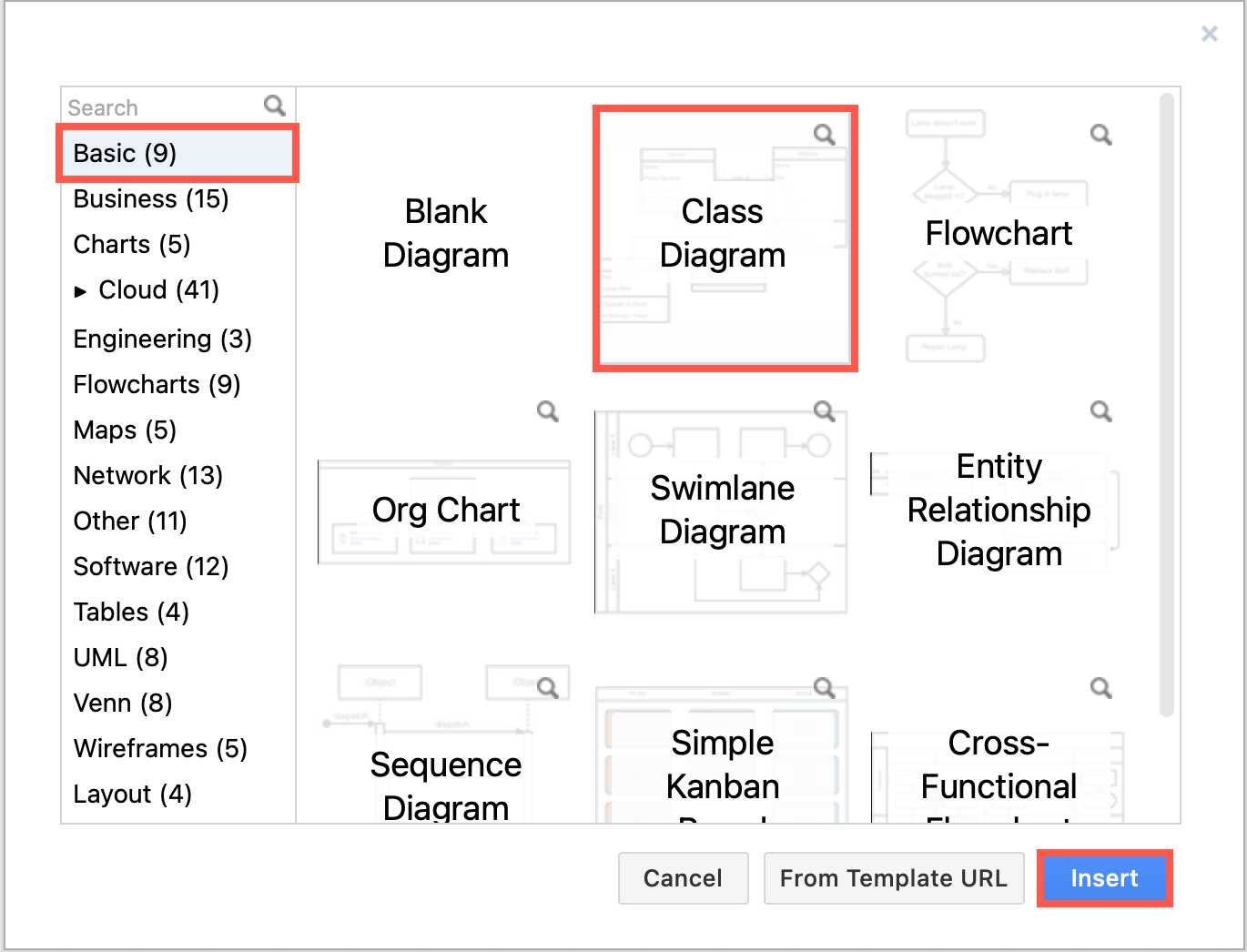


Figure 2.8. Selecting a class diagram

### Work with class shapes

UML class shapes always have a Class Name.

Classes in detailed diagrams have two additional sections for attributes and methods, and usually follow the UML notation, detailed in the sections below.

Detailed UML classes are separated into *three sections*  with the class name at the top.

The middle section contains *the attributes* (data or variables) of the class: attributeName: type

The lower section contains the methods (operations or functions) that the class can execute: methodName(parameterName: type): type

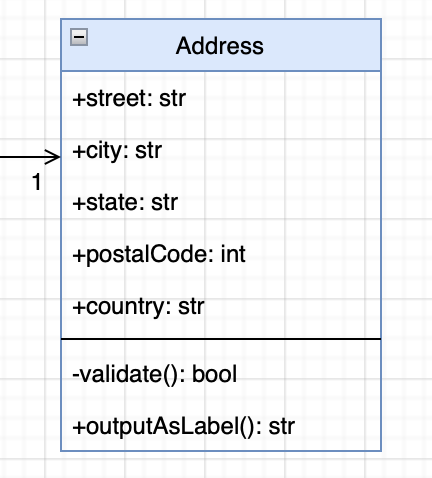


Figure 2.9 Address class having a name, attributes and methods

Parameter names, and data types are optional. They are typically used in a detailed class diagrams of an implementation or for formal specification documents.

### Editing UML classes in diagrams.net

**Select a row inside a class shape**: Click on a class shape once to select the entire shape. Click a second time, this time on a row inside the class, to select that attribute or method row.

**Rename a class**: Select the class and start typing to change its name.

**Edit an attribute or method**: Double click on the attribute or method to highlight all of the text. Start typing to replace it, or use the cursor keys or mouse to reposition the cursor within the text.

### Add an attribute or method

There are many ways to add new rows to class shapes in diagrams.net.

Add an attribute or divider from the library: Drag an Item 1 (attribute) or Divider shape from the UML shape library and drop it on an existing class shape on the drawing canvas in the position where you want to insert it.

**Clone a row**: Select a row, and press the keyboard shortcut Ctrl+Enter on Windows or Cmd+Enter on macOS to insert a clone of the selected row immediately below. The Ctrl+D/Cmd+D keyboard shortcut or right-clicking on the row and selecting Duplicate from the context menu work in the same way.

**Add a blank row**: In the Arrange tab of the format panel, click the Insert Row After or the Insert Row Above button. If you had selected the entire class shape, a blank row will be inserted at the end or at the start of your class, otherwise it will be inserted before or after the selected row.

**Move an attribute or method from another class**: Drag a row from one existing class and drop it on another.

### Attribute and method names in UML notation

Class names are usually in bold. Abstract class names are often written in italic.

Attribute and method names are typically written in lowercase, or camel case (forExample). Abstract methods are usually written in italic.

#### ****Visibility****

The visibility of attributes or methods is indicated with a symbol before their names.

+ **Public** elements are accessible from outside that class.

- **Private** elements are only accessible to methods inside that class.

# **Protected** elements are only accessible within its namespace (a group of classes and packages).

~ **Package** elements are only accessible within its package namespace (a group of related classes). Package diagrams are a specialised form of class diagram (Figure 2.10).

A forward slash / indicates an attribute is **derived** or computed from other attributes.

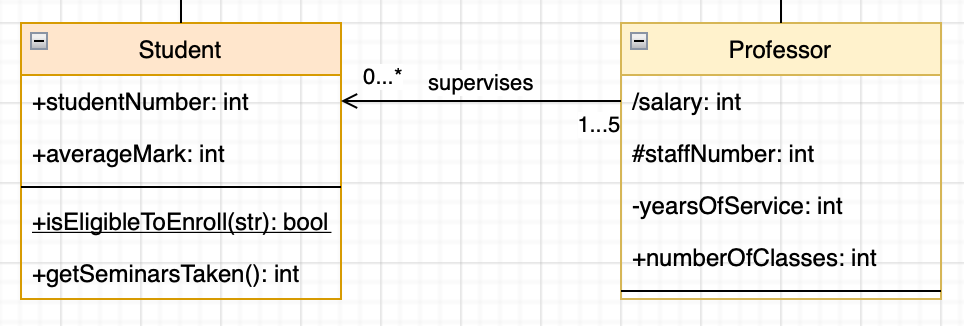


Figure 2.10 Class visibility

#### ****Scope****

**Instance scope** is assumed by default - attributes can have different values whenever that class is instanced, and the methods that are executed may alter that instance.

**Class scope** or static scope is shown by underlining the name of an attribute - attribute values are the same across all instances of that class.

#### ****Type****

You can add the data types to attributes and methods, including both the data passed to that method, and the data returned by it. Use a colon followed by the data type. For example: +name: string or +area(radius:float) :double

### Connectors for relationships in UML class diagrams

Define the type of relationship between classes using a connector that has a specific symbol, labels at either end, and either a solid or dashed line.

* **Inheritance (generalisation):** An open triangle at the parent class. This shows that the subclasses are specialisations of the parent class - they extend the parent class.
* **Implementation (realisation or execution):** A dashed line with an open triangle at the blueprint class. This is used where a class implements the functionality of a ‘blueprint’ class, and may be implemented differently depending on each class that implements it.
* **Dependency:** A dashed line with an open arrow. If the definition of class 2 changes, it will change class 1, but not the other way around (depends on).
* **Association:** A solid line between two classes. Add arrows at either end or both ends to show that the classes are aware (or unaware) of each other.
* **Aggregation:** An open diamond at class 1. A special type of association that shows that class 2 is a part of class 1.
* **Composition:** A solid diamond at class 1. A special type of aggregation that shows that class 2 cannot exist without class 1.

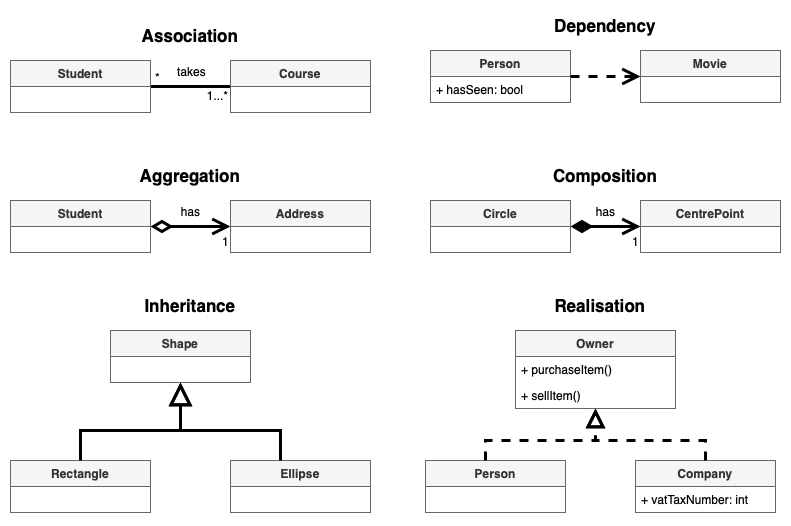


Figure 2.11 Connectors for relationships in UML class diagrams

#### Draw class relationships in diagrams.net

Click on or drag the connector you want to use from the UML shape library, then drag each end and drop it onto a class shape on the drawing canvas.

Alternatively, draw a connector from class 1 to class 2 or select an existing connector. Then change its source and target ‘arrows’ from the drop-down list in the Style tab of the format panel.

#### Connector labels

You can add labels to a connector to further define the relationship between the classes.

* Specify how many instances of each class are in the relationship: one to one, one to many, many to many, an exact number, etc.
* Explain why the relationship exists.
* Note ownership or roles.
* Clarify visibility.

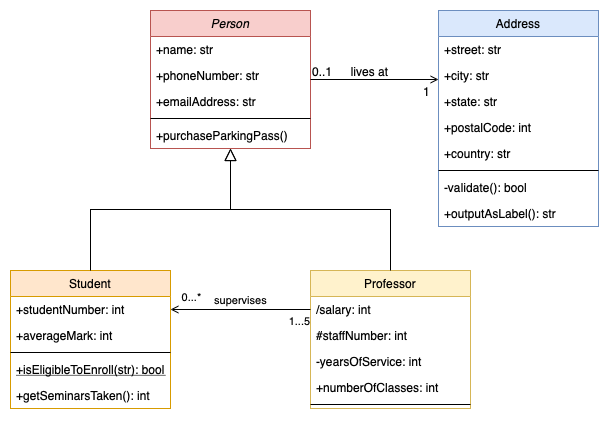


Figure 2.12 Connectors with labels

## The assignment

Using diagrams.net draw a UML class diagram that models the statements according to your option**[[1]](#footnote-1)**:

**Option #1.** A university is composed of one or more institutions and faculties. Every faculty contains departments, students and staff. Every faculty maintains an address book of entries for its staff.

**Option #2.** A software system type is social network. It allows to post messages with zero or more pictures, comment these messages, like or unlike messages. Also, it allows to store user accounts and their profiles.

**Option #3.** A software system is a file storage. It allows to add, rename, move and delete files and folders which are stored in different folders. Files could be of different types (video, audio, documents).

**Option #4.** A library contains books of one or more authors. Every Book has several copies (instances). Students are allowed to borrow one or more books (no more then 10). There is a log module which controls the borrowing process (who, when, what took and when he/she has to return books).

**Option #5.** A shop contains of several departments, each of them sells the product of different types. Every customer can make an order containing 1 or more items.

**Option #6.** There is a soccer (football) tournament. Several teams containing a couch, manager, doctor and players take place in the tournament. The teams play matches and the scores are stored.

**Option #7.** An image platform allows users to register and sign in. After that users can post images, add categories, tags and descriptions to them. Users can create and delete albums of any set of images.

**Option #8.** A polling system allows users to register and sign in. There are two types of users: admins and regular users. The admins can create polls, add questions and answers for them. The regular user can vote for the questions one or more times.

**Option #9.** A testing system allows users to register and sign in. There are two types of users: admins and regular users. The admins can create questions, type of questions, count of points for them, and add/edit answers for them including correct answers. The regular users are able to answer the questions and retrieve the count of points for correctly answered questions.

**Option #10.** A ticket service allows to create events containing title, date, performer name of the event. The event is performed in a concert hall. The regular users are allowed to order the tickets to these events.

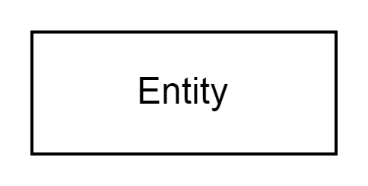
# Laboratory work #2. Database modeling and design

## Entity-Relational Diagrams (ERD)

The main goal of the database design stage is the formalization of data and their relationships in the subject area under consideration. The first stage of formalization or the transition from unstructured to structured data is the presentation of information in the form of one of the so-called information-logical models. The most used among such models is "entity-relationship" (ER, entity-relationship). It allows you to define the main entities, connections between entities and their attributes in an intuitively understandable graphic form. It should be noted that an important advantage of this model is the possibility of automatic transformation into a scheme of a real database, in particular, a relational one.

So, the entity-relationship model consists of the following elements.

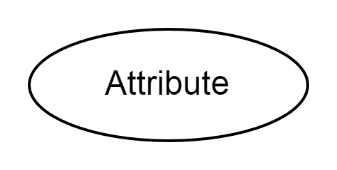
1. An entity that describes a real or imaginary class of objects in the subject field. For example, a student, a book, a computer, an order, etc. An instance of an entity is a specific representative of a class of objects, for example, student "Dmytrenko", book "C++", computer "Model M1", order "N321", etc. Graphically, the entity is indicated by a rectangle:

****

1.1. A weak entity, instances of which cannot exist in the subject field independently, but require the presence of another entity on which the given depends. For example, the entity "order item" cannot exist without the entity "order", "blog post" cannot exist without its author. It is indicated by a rectangle with a double border.

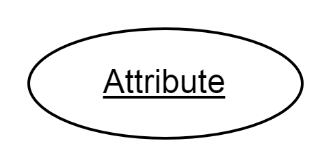
****

2. An attribute that defines a certain characteristic of an entity. For example, in the Student entity, the attributes are "record book number", "surname", "date of birth", etc. Graphically, the attribute is denoted as an oval:



In some specific cases, different types of attributes are used:

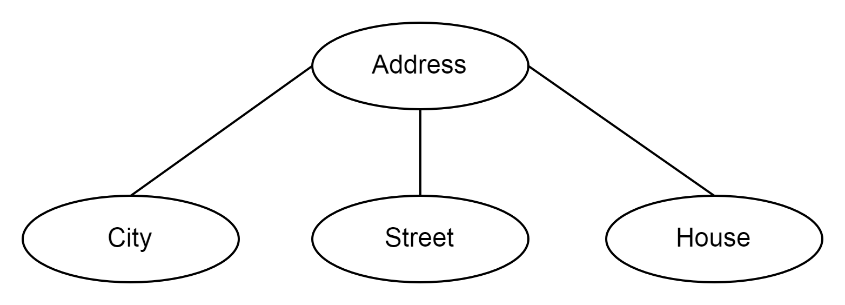
2.1. A key attribute intended to uniquely identify a specific instance of an entity. For example, the "Number of the student's record book" or "VIN number of the car". Note that any entity must have a key attribute or, in the case of a composite key, key attributes. It is graphically represented as a normal attribute with an underlined name:

****

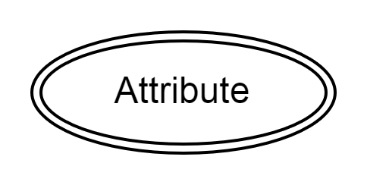
2.2. A derived attribute that is intended to represent an entity characteristic whose value depends on another attribute, such as the "discount price" of an item, is calculated based on the "base price" and "discount". It should be noted that saving derived attributes in databases is impractical in most cases, as it leads to saving redundant information. It is graphically displayed as follows:

****

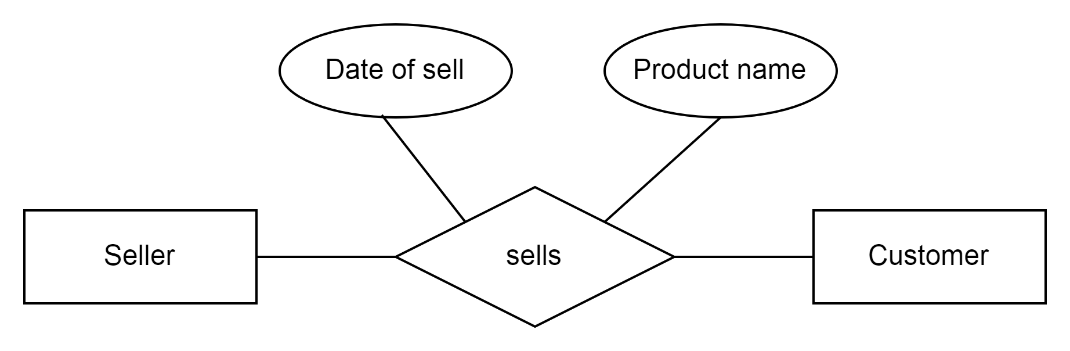
2.3. A composite (composite) attribute defines a complex characteristic of an entity, which also consists of a group of other attributes. An example of a composite attribute is an address consisting of a city, a street, a house number, and an apartment. It is graphically indicated as follows:

****

2.4. A multivalued attribute displays characteristics that can have multiple values ​​at the same time, for example, a person can have multiple email addresses or phone numbers. This attribute is denoted as follows:

****

**3.** Connectionbetween entities determines the degree of their relationship. The most common type of relationship is binary, which connects two entities. A connection is denoted as a rhombus connecting two entities with lines and, like entities, can have its own attributes. For example, the relation "sold" between the entities Seller and Buyer can have the attributes "Sold Date" and "Item Name". The following figure graphically illustrates this:



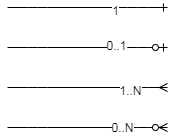
3.1. The cardinality of relationships is an important quantitative characteristic of the ratio of instances of entities. The following types of connections are distinguished:

**1:1**- to each other

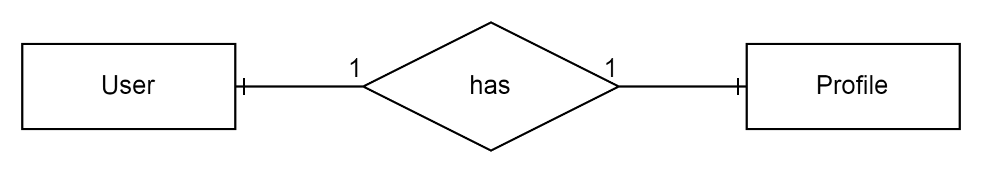
**1:N**- one to many

**N:M**- many to many

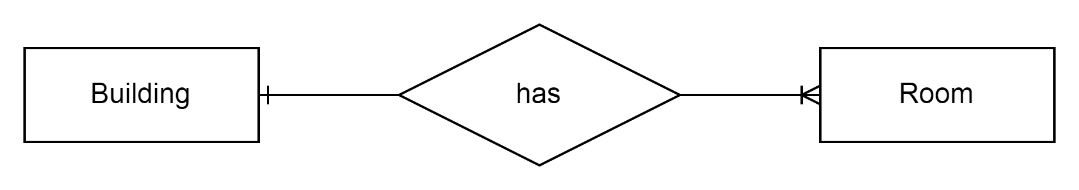
Graphically, the cardinality of connections is indicated by a special sign next to the end of the line:



At the same time, in the case when the connection is not mandatory, the symbol "0" is set. Similar special signs must be placed on both sides of the line as in the example:

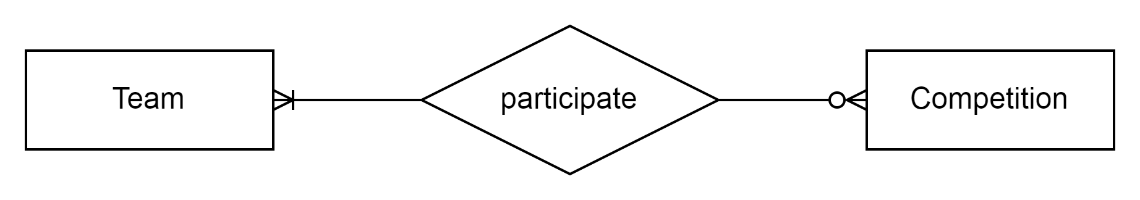


Such a scheme should be read as follows: "each user of the site has one profile and each profile on the site corresponds to exactly one user." The following example illustrates a one-to-many relationship:



At the same time, the educational building necessarily contains one or more classrooms, and one classroom is contained in one and only one building.

An example of using a many-to-many (N:M) relationship is illustrated in the following figure:



In the picture: one sports team participates in several competitions and several teams participate in one competition.

### Example of Entity-Relationship Diagram

The idea of an example is to create a conceptual model of data for polling application. The idea is to store user accounts of two types: administrators and regular users. Administrators can create a poll, enter some questions and possible options. On the other hand, regular users can enter answers to these questions.

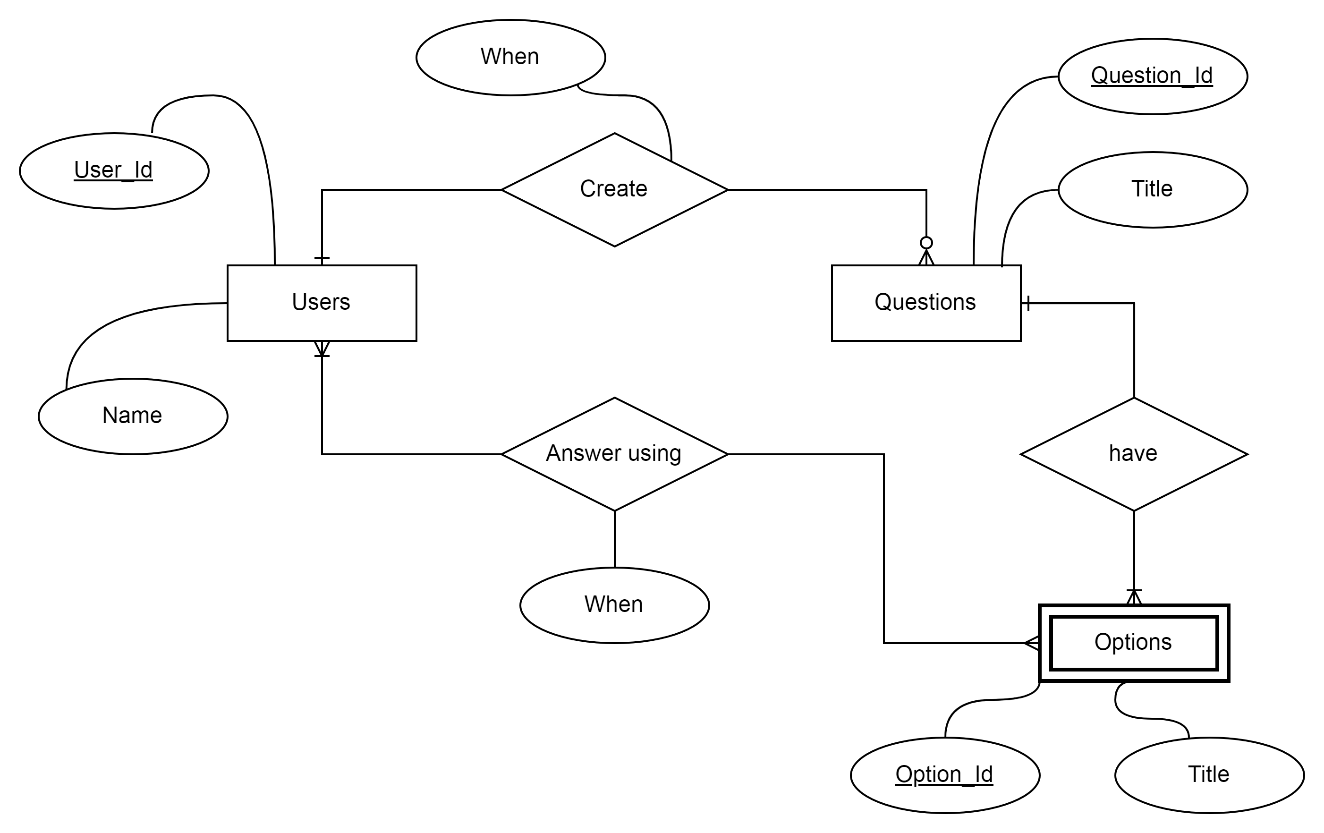


Figure 3.1 Polling ERD

The diagram at the Figure 3.1 has four entities: users, questions, options and answers. There are some relationships between them. For example, users can select options to the questions. There are also some different types of cardinality: 1:N and N:M.

### The algorithm of translation ERD to relational database schema

Since ER diagram gives us the good knowledge about the requirement and the mapping of the entities in it, we can easily convert them as tables and columns. i.e.; using ER diagrams one can easily create relational data model, which nothing but the logical view of the database.

Follow the steps given below for the conversion of the ER diagrams to tables in the database management system (DBMS).

**Step 1** − *Conversion of strong entities*

For each strong entity create a separate table with the same name.

Includes all attributes, if there is any composite attribute divided into simple attributes and has to be included. Ignore multivalued attributes at this stage. Select the primary key for the table.

**Step 2** − *Conversion of weak entity*

For each weak entity create a separate table with the same name. Include all attributes. Include the primary key of a strong entity as foreign key is the weak entity. Declare the combination of foreign key and decimator attribute as P key from the weak entity.

**Step 3** − *Conversion of one-to-one relationship*

For each one to one relation, say A and B modify either A side or B side to include the primary key of the other side as a foreign key. If A or B is having total participation, then that should be a modified table. If a relationship consists of attributes, include them also in the modified table.

**Step 4** − *Conversion of one-to-many relationship*

For each one to many relationships, modify the M side to include the primary key of one side as a foreign key. If relationships consist of attributes, include them as well.

**Step 5** − *Conversion of many-many relationship*

For each many-many relationship, create a separate table including the primary key of M side and N side as foreign keys in the new table. Declare the combination of foreign keys as P for the new table. If relationships consist of attributes, include them also in the new table.

**Step 6** − *Conversion of multivalued attributes*

For each multivalued attribute create a separate table and include the primary key of the present table as foreign key. Declare the combination of foreign key and multivalued attribute as primary keys.

**Step 7** − Conversion of n-ary relationship

For each n-ary relationship create a separate table and include the primary key of all entities as foreign key. Declare the combination of foreign keys as primary key.

Let’s consider the ER diagram from the Figure 3.1.

The basic rule for converting the ER diagrams into tables is *to convert all the Entities in the diagram to tables*.

All the entities represented in the rectangular box in the ER diagram become independent tables in the database. In the diagram, USERS, QUESTIONS and OPTIONS forms individual tables.

*All single valued attributes of an entity are converted to a column of the table*.

All the attributes, whose value at any instance of time is unique, are considered as columns of that table. In the USER Entity, *name* forms the columns of USER table. Similarly, *Title* form the columns of QUESTIONS table. And so on.

*Key attribute in the ER diagram becomes the Primary key of the table.*

In diagram above, User\_id, Question\_Id and Option\_id are the key attributes of the entities. Hence, we consider them as the primary keys of respective table.

## Database design using pgAdmin 4

pgAdmin 4 a very popular open source platform fully dedicated to PostgreSQL and has a graphical user interface administration tools to manage your relational databases. Some features include a query tool for SQL statements and importing/exporting csv files.

The Entity-Relationship Diagram (ERD) tool is a database design tool that provides a graphical representation of database tables, columns, and inter-relationships. ERD can give sufficient information for the database administrator to follow when developing and maintaining the database. The ERD Tool allows you to:

* Design and visualize the database tables and their relationships.
* Add notes to the diagram.
* Auto-align the tables and links for cleaner visualization.
* Save the diagram and open it later to continue working on it.
* Generate ready to run SQL from the database design.
* Generate the database diagram for an existing database.
* Drag and drop tables from browser tree to the diagram.

PgAdmin will use your preferred web browser to display a graphical user interface. You don’t need internet to view local servers. It will prompt you for a master password every time you open pgAdmin to get access. After getting access click Servers(1) on the left side to open up your PostgreSQL 12 server. If you don’t see a server, try restarting pgAdmin.

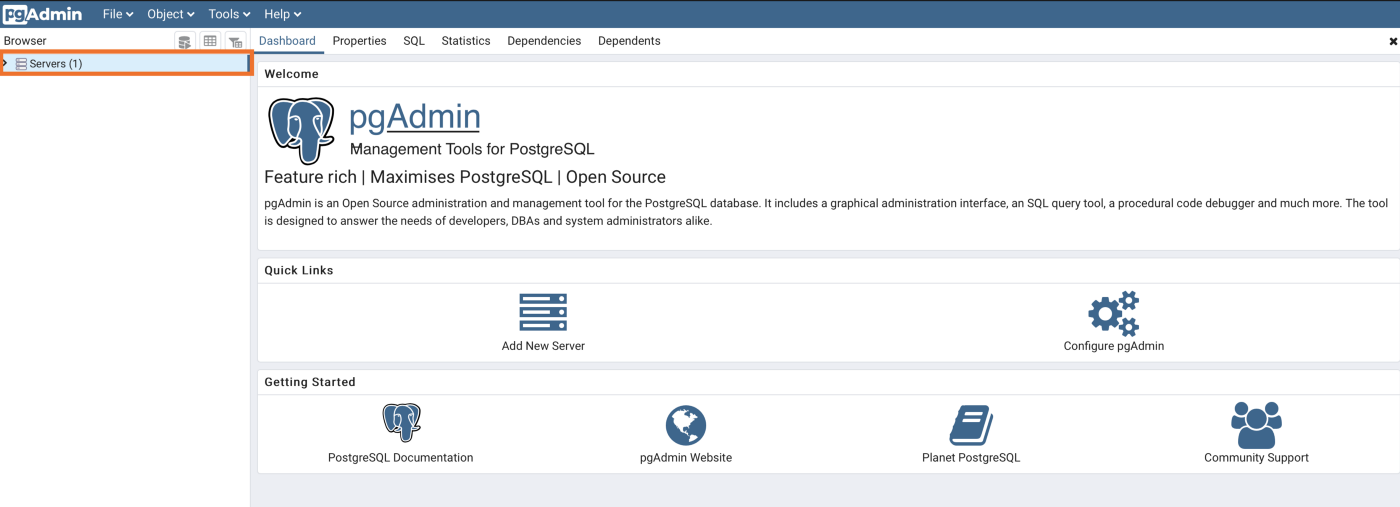


Figure 3.2 Main window of pgAdmin 4 tool.

After you open up your database you’ll get a tree view menu according to the picture below. You will see a database named **postgres**. Selecting the database will bring up an activity dashboard to view traffic information (Figure 3.3).

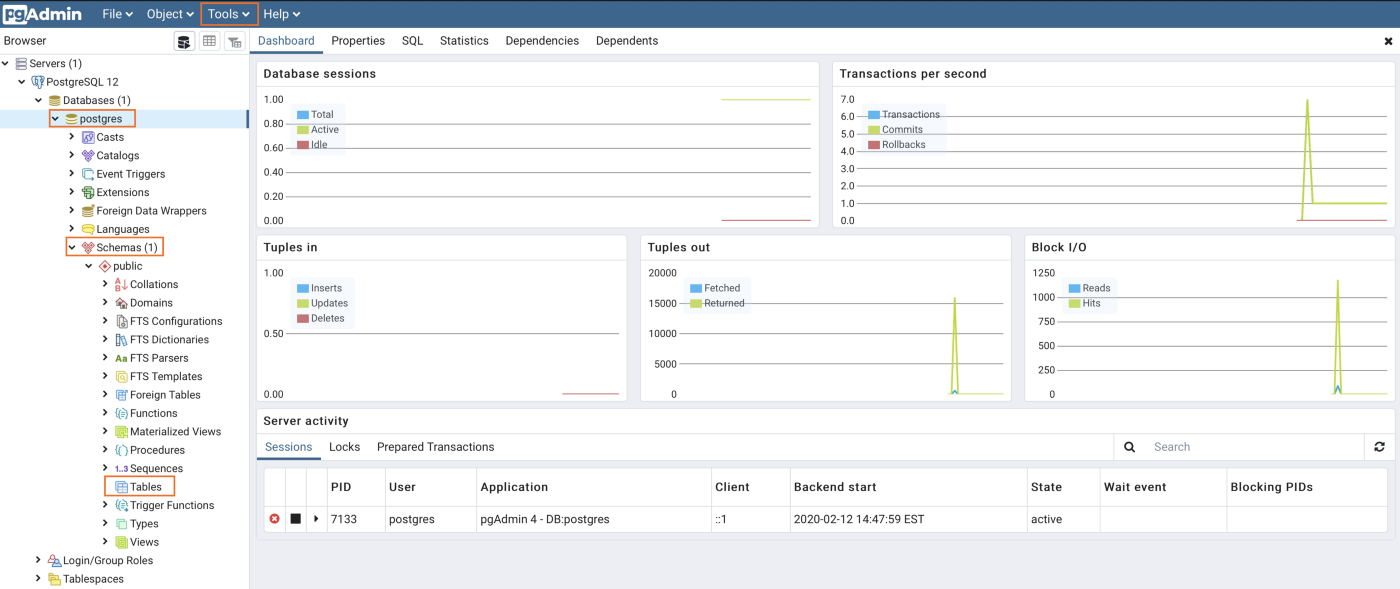


Figure 3.3 A tree of objects of Postgresql

You have access to useful tools by right-clicking on any object in the tree view menu. We can **create**a new database, schema, and tables. As well as viewing individual table data and customizing an existing table.

### ****Create Table Example****

Let’s create a table by right clicking on **Tables**and click**Table…**

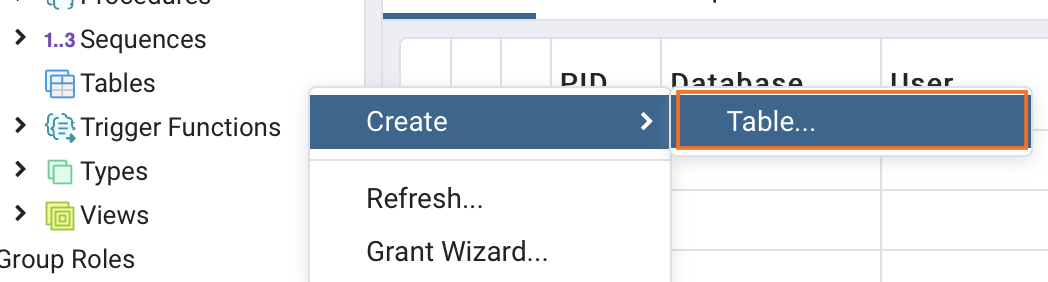


Figure 3.4 Creation of a table

Give your table a **name** and then click **Columns**

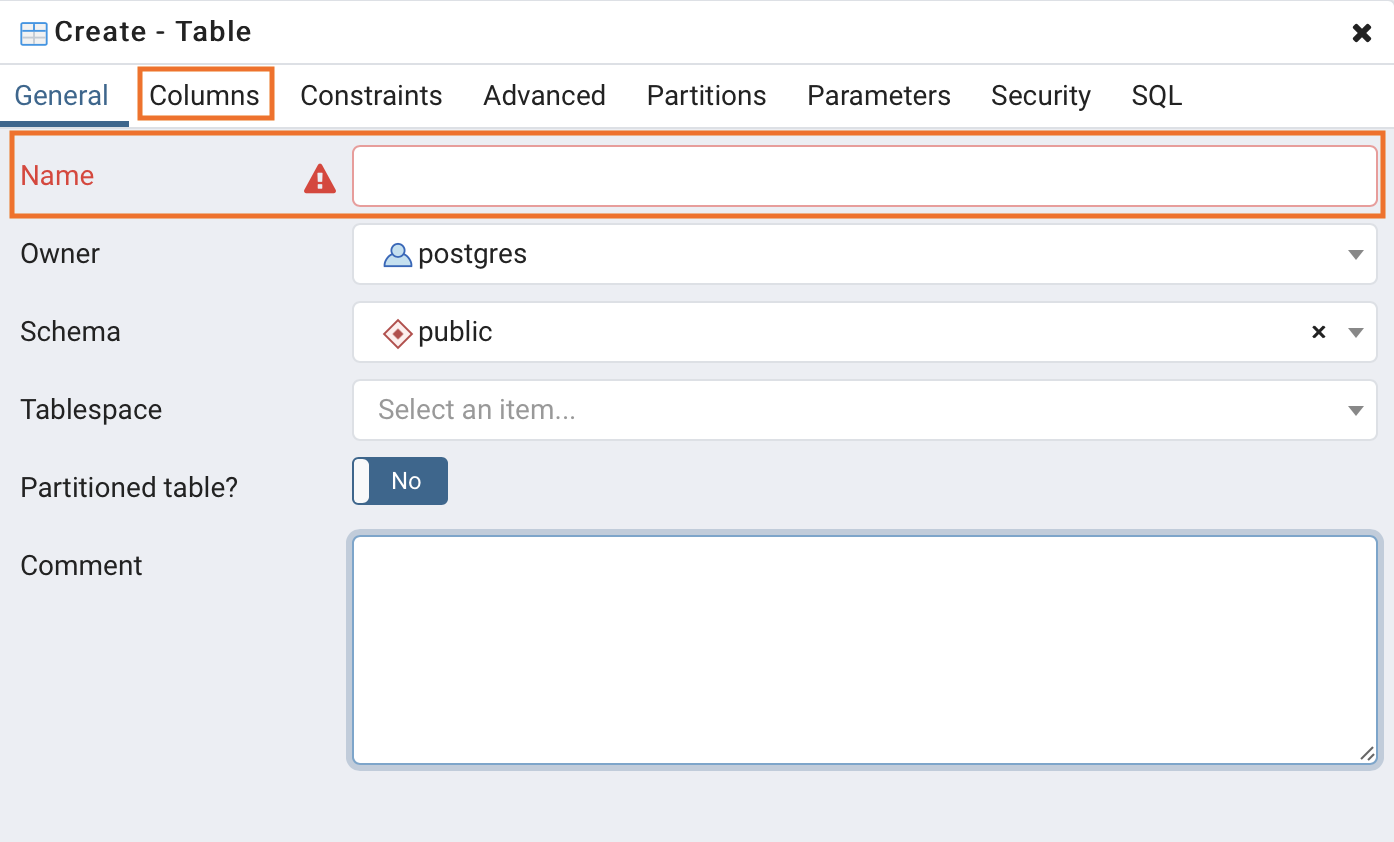


Figure 3.5 Entering column data

Click the **+** symbol to add columns

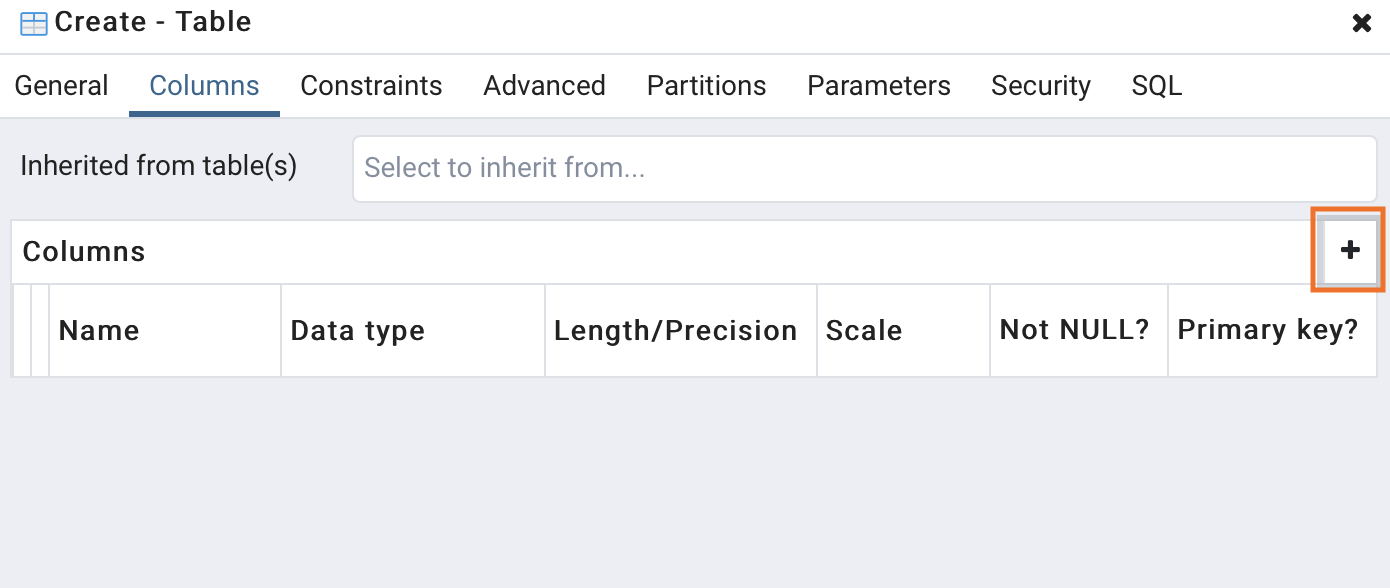


Figure 3.6 Insertion of a column

**Name** your column, select the **data type**, and give a **length** if needed. Then click **Save**

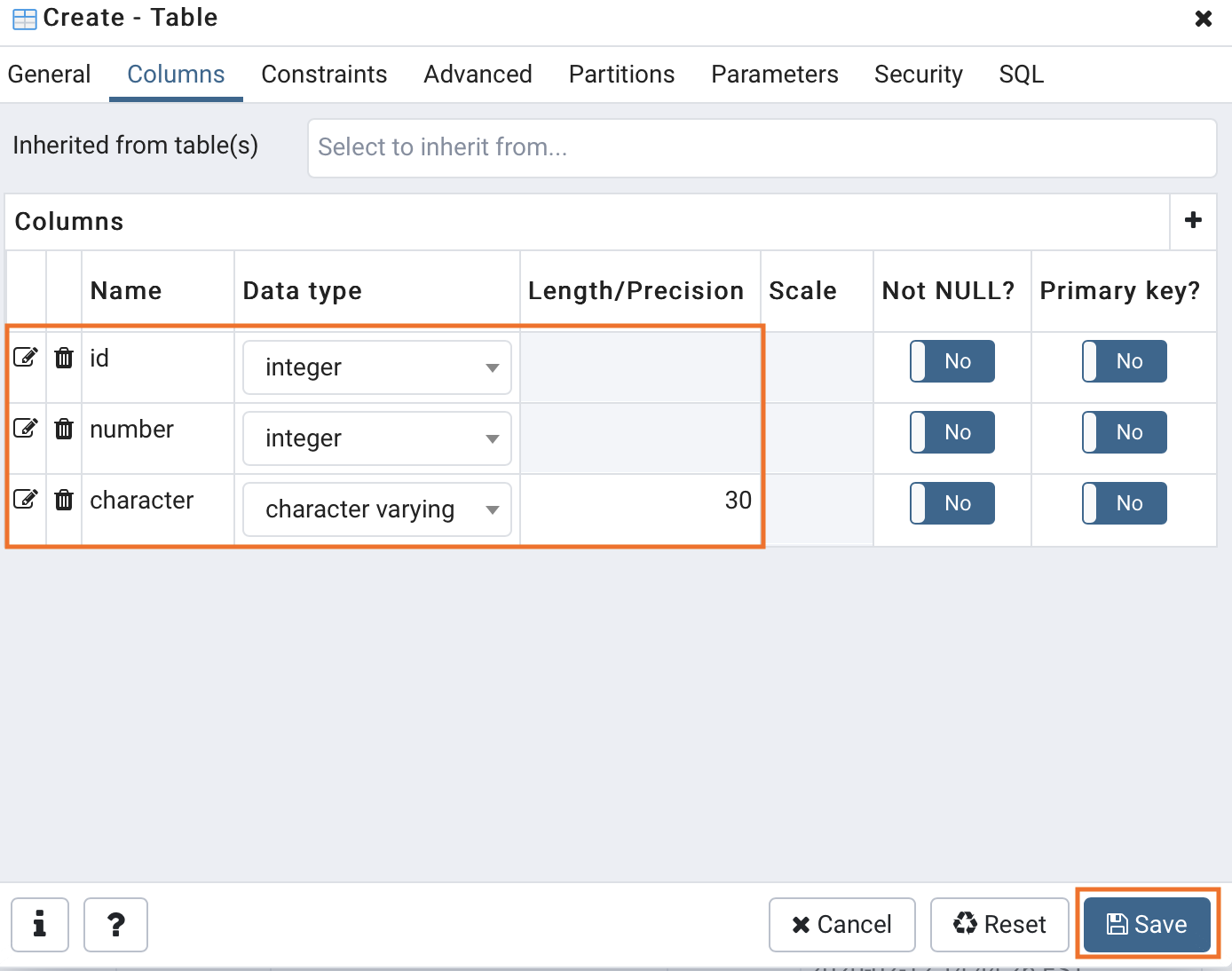


Figure 3.7 Saving columns data

Now that you have created a table, view it under the **Tables** object. Right click and refresh if it didn’t update.

### ****Query Tool Example****

When an object is selected under the database tree view menu, you can click the **Tools** tab and click **Query Tool**. Query tool brings up an editor to execute SQL statements. You can also right click the database and click **Query Tool …**

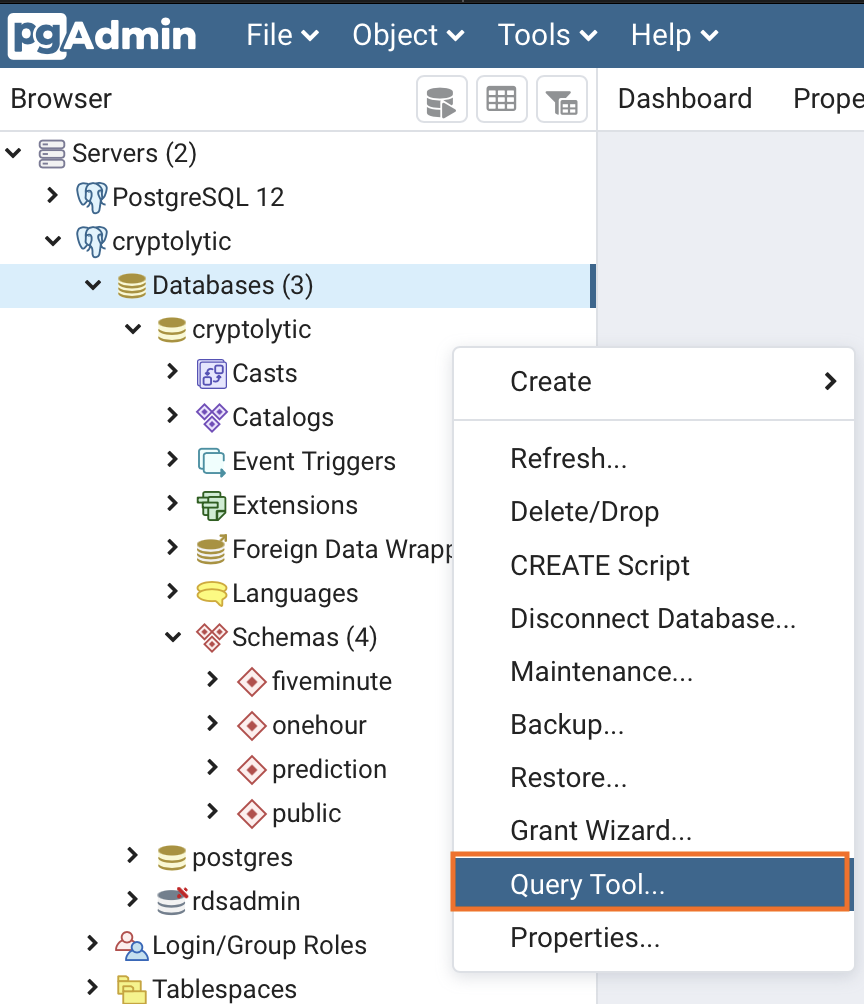


Figure 3.8 A Query tool

A tab called **query editor** will open up on the right. You can write SQL statements and click the **play button** to execute. Your results will show below the editor on the **Data Output** tab.

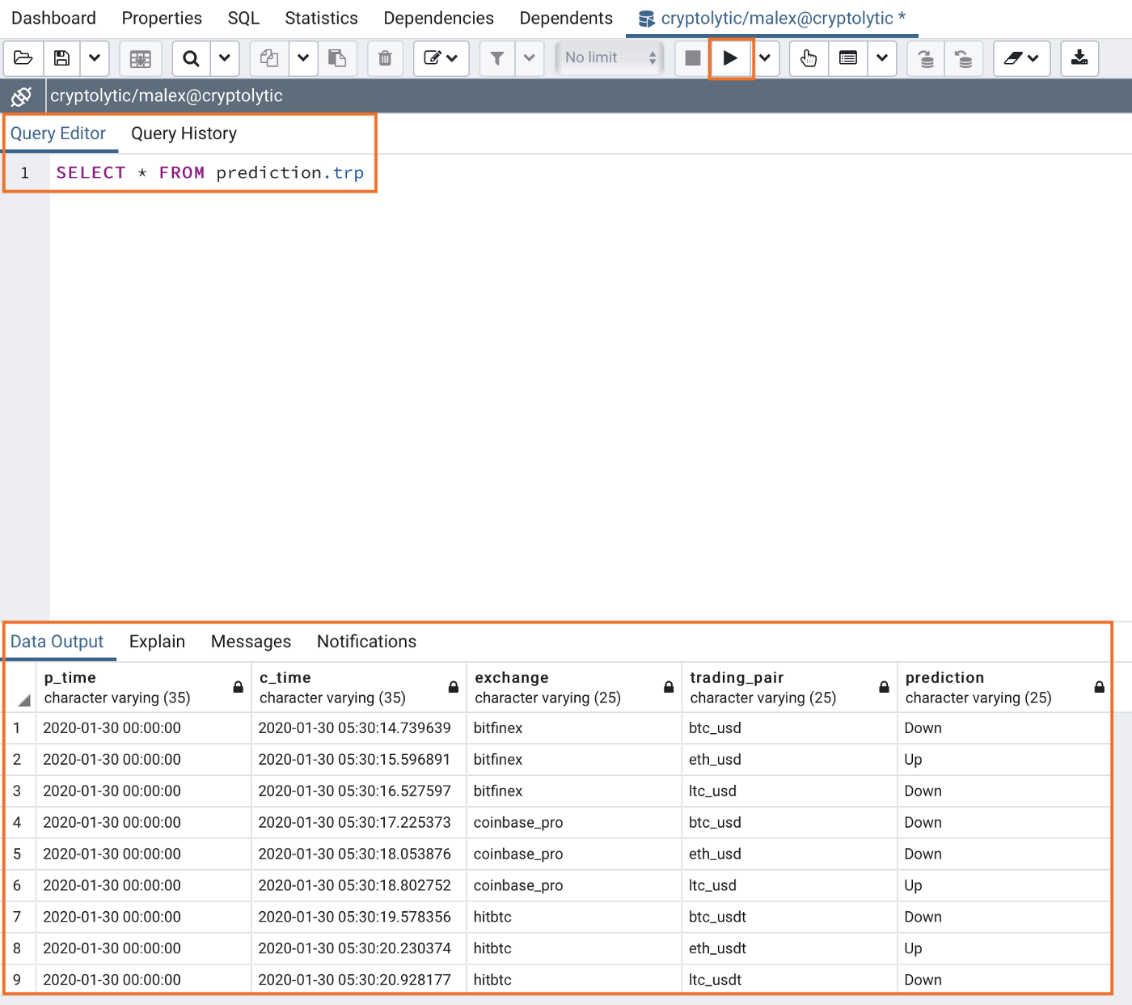


Figure 3.9 A query execution results

### pgAdmin 4 ERD Tool

The Entity-Relationship Diagram (ERD) tool is a database design tool that provides a graphical representation of database tables, columns, and inter-relationships. ERD can give sufficient information for the database administrator to follow when developing and maintaining the database. The ERD Tool allows you to:

* Design and visualize the database tables and their relationships.
* Add notes to the diagram.
* Auto-align the tables and links for cleaner visualization.
* Save the diagram and open it later to continue working on it.
* Generate ready to run SQL from the database design.
* Generate the database diagram for an existing database.
* Drag and drop tables from browser tree to the diagram.

#### Toolbar[¶](https://www.pgadmin.org/docs/pgadmin4/6.9/erd_tool.html#toolbar)

The ERD Tool toolbar uses context-sensitive icons that provide shortcuts to frequently performed tasks. The option is enabled for the highlighted icon and is disabled for the grayed-out icon.

ERD tool toolbar

Figure 3.10 ERD toolbar

Hover over an icon on Toolbar to display a tooltip that describes the icon’s functionality.

#### Table Dialog

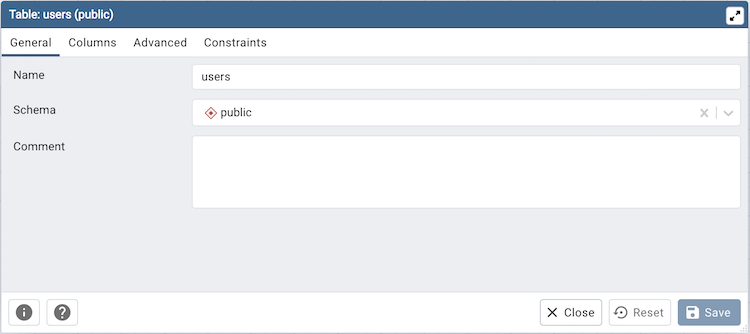


Figure 3.11 Table dialog

The table dialog allows you to:

* Change the table structure details.
* It can be used edit an existing table or add a new one.
* Refer table dialog for information on different fields.

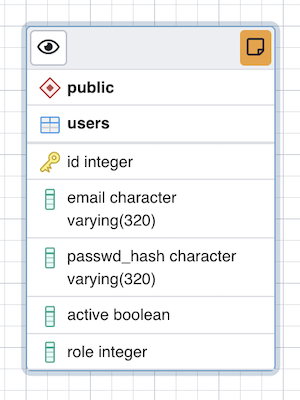


Figure 3.12 Table attributes

The table node (Figure 3.12) shows table details in a graphical representation:

* The top bar has a *details toggle button* that is used to toggle column details visibility. There is also a *note button* that is visible only if there is some note added. you can click on this button to quickly change the note.
* The first row shows the schema name of the table. Eg. *public* in above image.
* The second row shows the table name. Eg. *users* in above image.
* All other rows below the table name are the columns of the table along with data type. If the column is a primary key then it will have lock key icon eg. id is the primary key in above image. Otherwise, it will have column icon.
* you can click on the node and drag to move on the canvas.
* Upon double click on the table node or by clicking the edit button from the toolbar, the table dialog opens where you can change the table details. Refer table dialog for information on different fields.

#### The One to Many Link Dialog

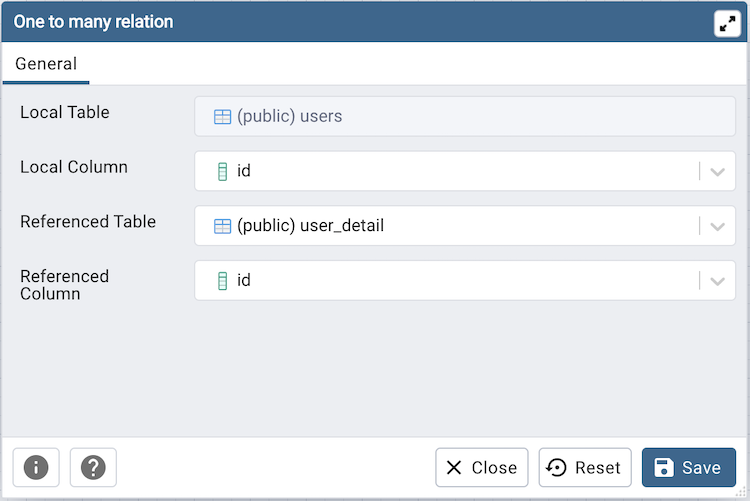


Figure 3.13 One to many relation

The one to many link dialog allows you to:

* Add a foreign key relationship between two tables.
* *Local Table* is the table that references a table and has the *many* end point.
* *Local Column* the column that references.
* *Referenced Table* is the table that is being referred and has the *one* end point.
* *Referenced Column* the column that is being referred.

#### The Many to Many Link Dialog

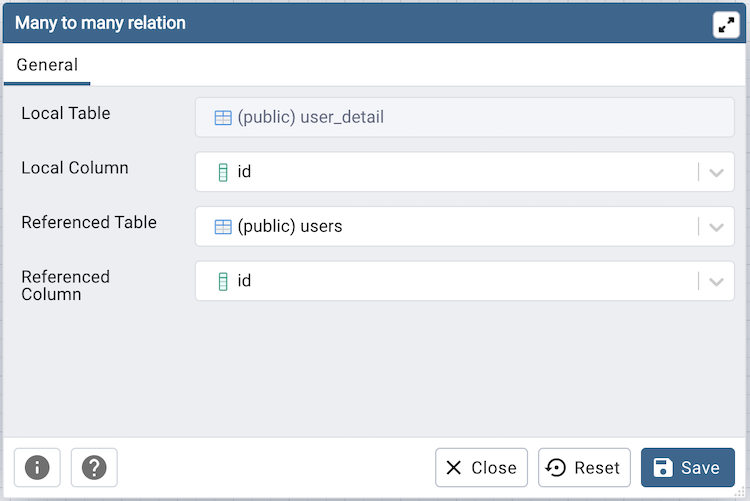


Figure 3.14 Many to many relation

The many to many link dialog allows you to:

* Add a many to many relationship between two tables.
* It creates a relationship tables having columns derived from the two tables and link them to the tables.
* *Left Table* is the first table that is to be linked. It will receive the *one* endpoint of the link with the new relation table.
* *Left Column* the column of the first table, that will always be a primary key.
* *Right Table* is the second table that is to be linked. It will receive the *one* endpoint of the link with the new relation table.
* *Right Column* the column of the second table, that will always be a primary key.

#### The Table Link

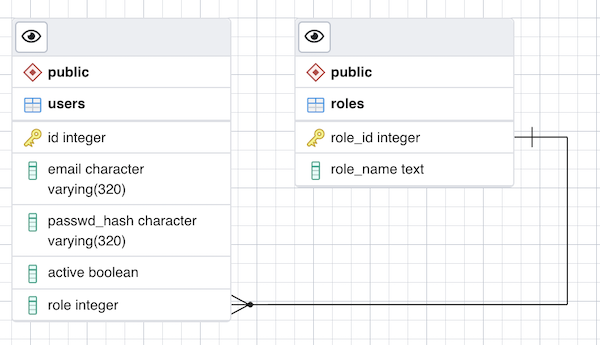


Figure 3.15 The table link

The table link shows relationship between tables:

* The single line endpoint of the link shows the column that is being referred.
* The three line endpoint of the link shows the column that refers.
* If one of the columns that is being referred or that refers is removed from the table then the link will get dropped.
* you can click on the link and drag to move on the canvas.

## The assignment

Taking into account your domain area from The assignment of laboratory work #1, do the next tasks:

1. Using diagrams.net create an entity-relationship diagram
2. Using pgAdmin 4 create a database schema

# Laboratory work #3. Software design patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

All patterns can be categorized by their intent, or purpose. This book covers three main groups of patterns:

* *Creational patterns* provide object creation mechanisms that increase flexibility and reuse of existing code.
* *Structural patterns* explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
* *Behavioral patterns* take care of effective communication and the assignment of responsibilities between objects.

## Creational design patterns

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

[**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
Creates an instance of several families of classes

[**Builder**](https://sourcemaking.com/design_patterns/builder)  
Separates object construction from its representation

[**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)Creates an instance of several derived classes

[**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

[**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
A fully initialized instance to be copied or cloned

[**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
A class of which only a single instance can exist

## Structural design patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

**Adapter**

Match interfaces of different classes

**Bridge**

Separates an object’s interface from its implementation

**Composite**

A tree structure of simple and composite objects

**Decorator**

Add responsibilities to objects dynamically

**Facade**

A single class that represents an entire subsystem

**Flyweight**

A fine-grained instance used for efficient sharing

**Private Class Data**

Restricts accessor/mutator access

**Proxy**

An object representing another object

## Behavioral design patterns

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

**Chain of responsibility**

A way of passing a request between a chain of objects

**Command**

Encapsulate a command request as an object

**Interpreter**

A way to include language elements in a program

**Iterator**

Sequentially access the elements of a collection

**Mediator**

Defines simplified communication between classes

**Memento**

Capture and restore an object's internal state

**Null Object**

Designed to act as a default value of an object

**Observer**

A way of notifying change to a number of classes

**State**

Alter an object's behavior when its state changes

**Strategy**

Encapsulates an algorithm inside a class

**Template method**

Defer the exact steps of an algorithm to a subclass

**Visitor**

Defines a new operation to a class without change

1. Your option is the last digit of your student ID card number + 1. [↑](#footnote-ref-1)