Complex Systems Theory and Practice

Lecture 5

Piotr Syga

2021-11-05

Authorization – recall

Here we assume, for simplicity, that the password is given in plaintext – do not ever keep passwords in plaintext.

Fixed user, fixed password

```
SELECT userid FROM users WHERE name='Bob' AND password='BobbyB';
```

Authorization – recall

Here we assume, for simplicity, that the password is given in plaintext – do not ever keep passwords in plaintext.

Fixed user, fixed password

```
SELECT userid FROM users WHERE name='Bob' AND password='BobbyB';
```

Using procedures

```
CREATE PROCEDURE auth (IN usr char(50), IN pass char(50), OUT ID
INT)
  SELECT userID INTO ID FROM users WHERE name=usr AND
password=pass;
```

How to use it: `CALL auth ('Bob','BobbyB',@uid)` – not very user friendly, hence we do move the quotation marks to the query.

Using concatenation

```
SET @str=CONCAT("SELECT userID FROM users WHERE name='",@usr," '
AND password='",@pass," ';");
```

Learning the database

Just the same way we can learn the contents of the database, e.g.:

- Does any username start with 'A':
  Password - `' OR EXISTS(SELECT * FROM users WHERE name LIKE 'A%') AND ''='`
- Is there a user with the same password as Bob:
  Password - `' OR EXISTS(SELECT * FROM users uB JOIN users u WHERE uB.name='Bob' AND u.password=uB.password AND u.name<>uB.name) AND ''='`
- Do we have more than 1000 users:
  Password - `' OR (SELECT COUNT(*) FROM users)>1000 AND ''='`
- Does the database *SecretProject* include *users* table:
  Password - `' OR EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA='SecretProject' AND TABLE_NAME='users') AND ''='`

Learning the database

Just the same way we can learn the contents of the database, e.g.:

- Does any username start with 'A':
  Password - `' OR EXISTS(SELECT * FROM users WHERE name LIKE 'A%') AND ''='`
- Is there a user with the same password as Bob:
  Password - `' OR EXISTS(SELECT * FROM users uB JOIN users u WHERE uB.name='Bob' AND u.password=uB.password AND u.name<>uB.name) AND ''='`
- Do we have more than 1000 users:
  Password - `' OR (SELECT COUNT(*) FROM users)>1000 AND ''='` Does every user has a unique password?
- Does the database *SecretProject* include *users* table:
  Password - `' OR EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA='SecretProject' AND TABLE_NAME='users') AND ''='`

Blind attack

Starting knowledge

`SELECT` *columns* `FROM` *table* `WHERE` *col*=`'textfield';`

1. W have the check if the textfield is vulnerable – can we use ' we used extensively: **Ronnie O'Sullivan** – user unknown vs syntax error
2. Checking how the logging in reacts on: **user' OR "='** – email (pm) vs user unknown vs error
3. Learning columns: **u' AND (email IS NULL) AND "='** or **u' AND email IS NULL; - -**
   typical columns to check: **name**, **id**, **lastname**, **fullname**, **login**, **password**, ...

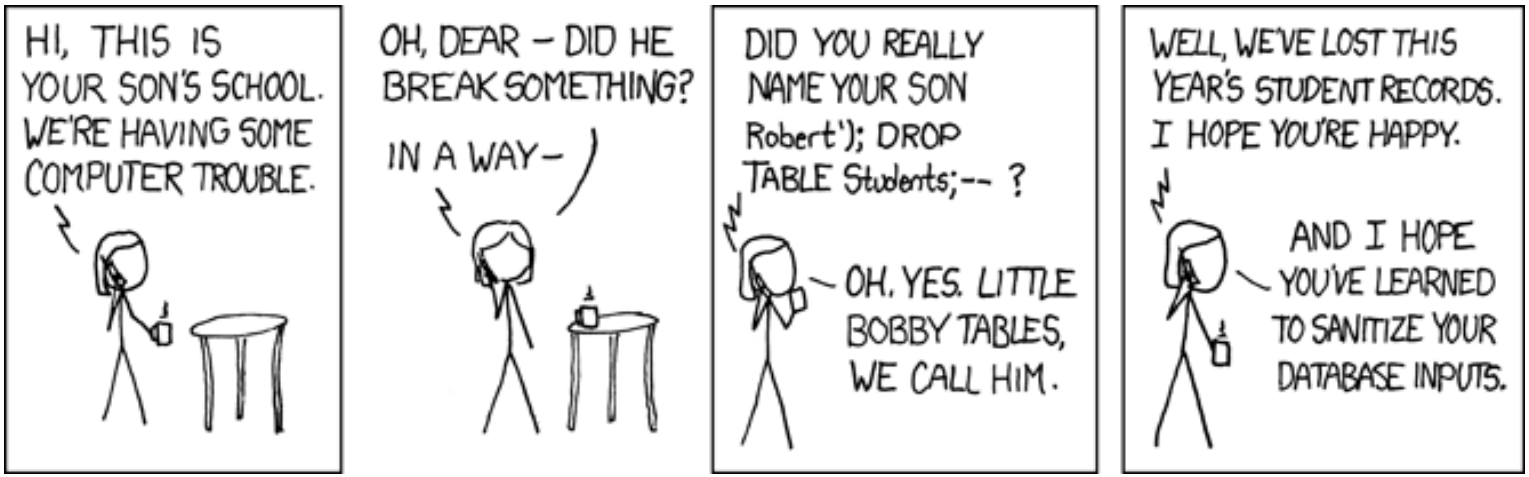
Blind attack

4. Learning the tables: **u' AND (SELECT COUNT(*) FROM users)>0; - -**
   typical tables to check:**accounts**, **logs**, **history**, ...
5. We verify if `users` table is used in the authorization: **u' AND users.name IS NULL; - -**
6. We check whether given user exists: **u' OR name LIKE '%Bob%**, we may also try our luck with passwords.
7. We can verify if we can add a new user: **u'; INSERT INTO users ('login','email','passwd','name')**
   VALUES ('Rogue','Daemon@Caraxes.ws','heir','Daemon'); - -
8. ...or maybe at least modify a user to a new one: **u'; UPDATE users**
   SET email = 'Daemon@Caraxes.ws'
   WHERE email LIKE '%@Caraxes.ws

Blind attack

and after we are bored . . .

9. . . . we can try out further privilages: **u'; DROP TABLE users; - -**



Prevention

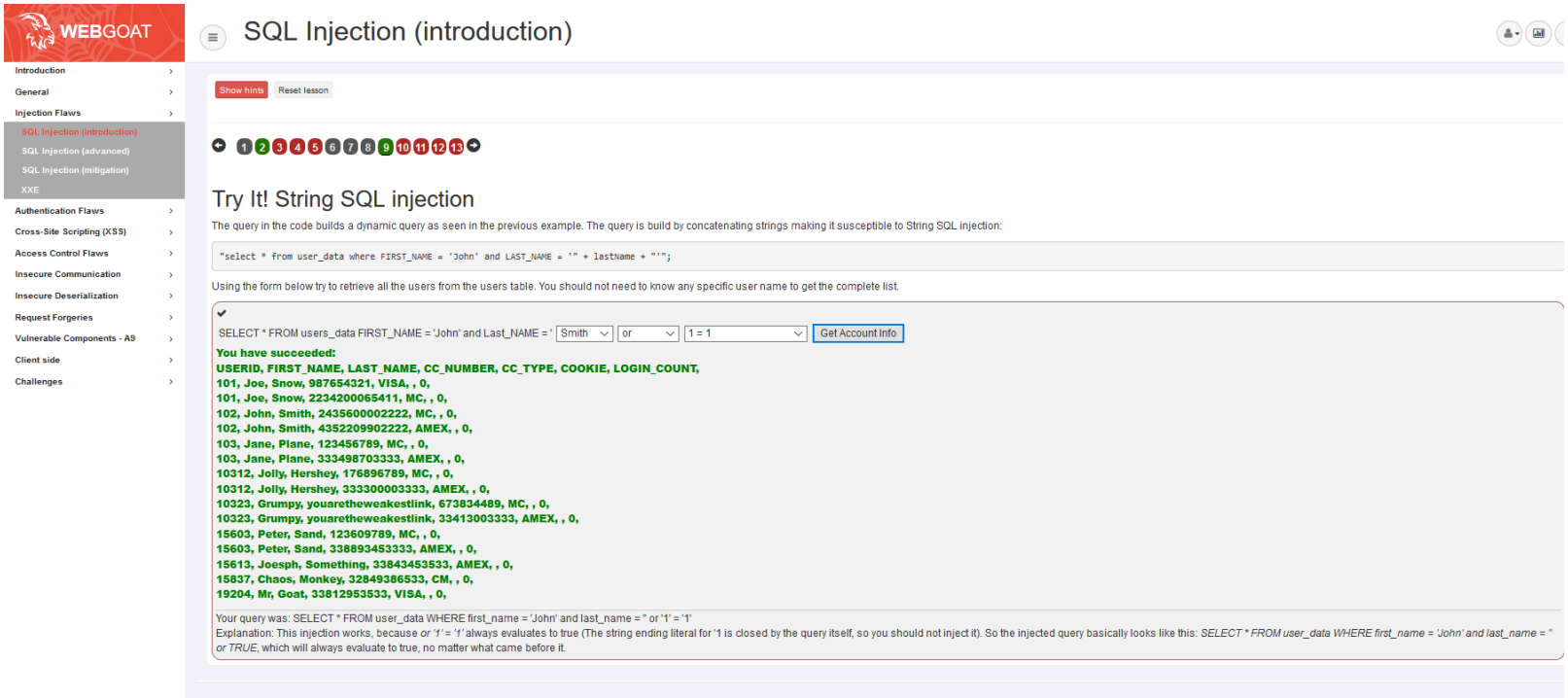
- Data sanitization: *white list* vs *black list*
  Note: **O'Sullivan** vs . . . WHERE id = 1000 OR 1=1
- Escaping: . . . \'; DROP TABLE users; - -
- Prepared statements and parameter management:
  Statement stmt = connection.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT id FROM users WHERE name = "+ txtfield); // **'; DROP TABLE users; - -**
  zmieniamy na:
  PreparedStatement stmt = connection.prepareStatement("SELECT id FROM users WHERE name = ?");
  stmt.setString(1, txtfield);
  ResultSet rs = ps.executeQuery();
- Access Level Control
- Utiling procedures and validating inputs before using them
- A proper reaction to abnormal behavior, configuring error messages

A useful Oracle Tutorial may be found here.

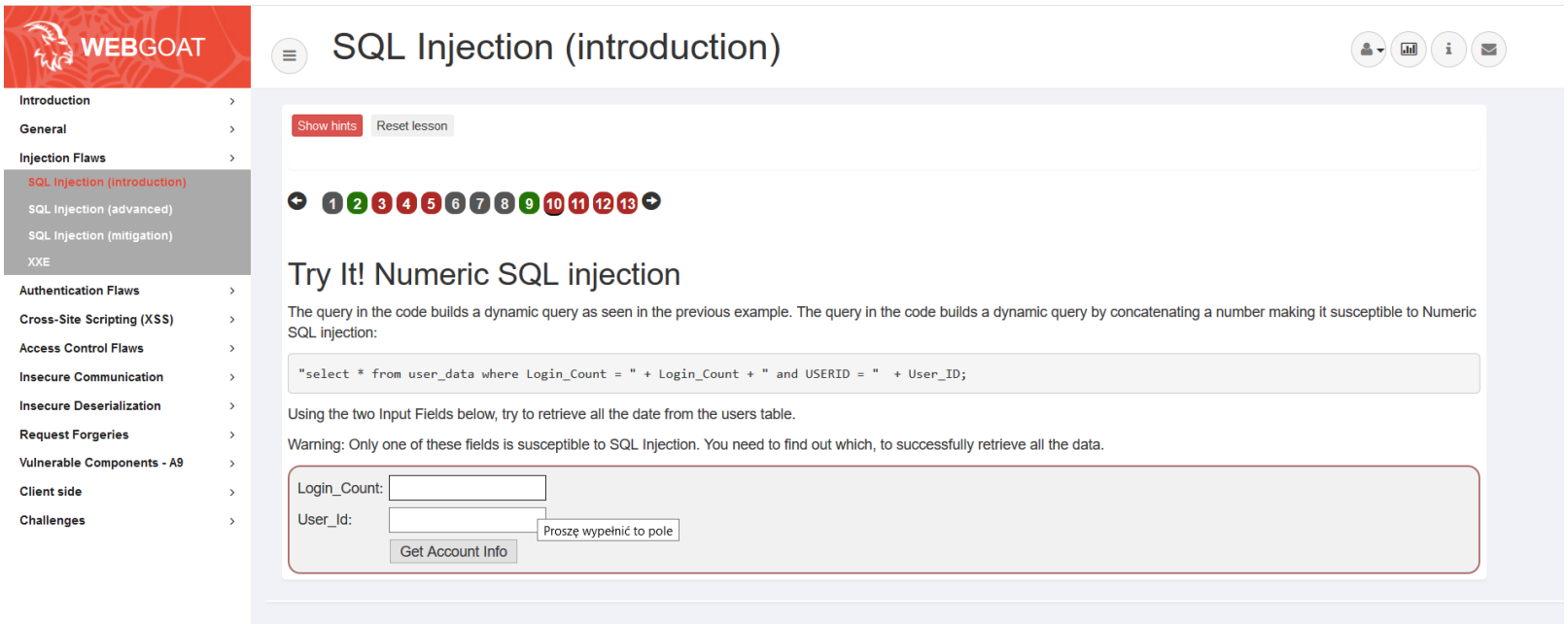
Testing environment – WebGoat

- Project that is, on purpose, vulnerable to SQL-injection, guiding the user through some potential attacks
- It mimic real-life systems that one may encounter or design
- Explain the vulnerability and asks you to perform simple tasks to test it
- Provides hints on attack prevention

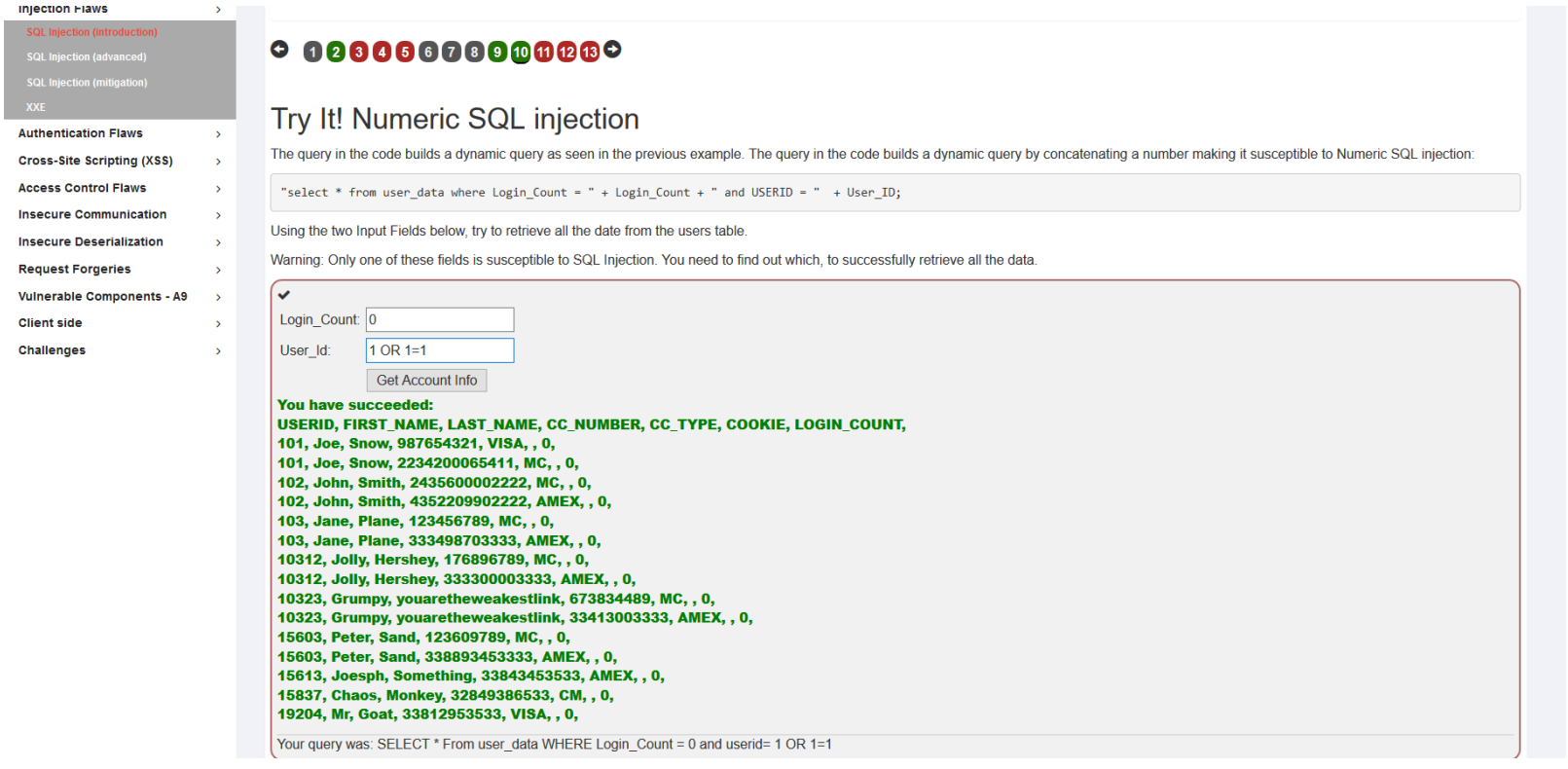
Testing environment – WebGoat



Testing environment – WebGoat



Testing environment – WebGoat



Finally, how to store credentials

We explained that the passwords should not be kept as a plaintext, so how do we store them

- if we keep the passwords outside of database, the authorization has to have access to that place
- if we encrypt them, the database-app has to have access to the encryption keys
- simpler way – hashing – during the verification we compare hashes instead of the passwords
- however, we do not want to simply hash the password – if we do so, all the passwords that are the same, will have the same hash (what if the attacker learns some passwords or has accounts in the app?)
- a random salt is added to the hash, the value of the salt is kept in the database

How to perform the authorization procedure in such a schema? Consider 3 actors: the user, the app, the database.

Planning out the database

- the database part of an app is designed in a similar way to high-level designing of the app
- we have to learn the user stories, specific functionalities that are required, main purpose of the database and the type and volume of the most common operations (e.g., OLTP vs. OLAP will determine the indexes and triggers required and will help with normalization concepts)
- we idenfity the entitites in the database, the relations between them
- we have to assign priorities to each entity and database component, as well as to extract the key functionalities
- we take into the account the restrictions of the project and efficiency, hardware or software constraints

Conceptual data model

At this stage we map domain-specific concepts to database components (entities). We identify business requirements. We start with identifying the main component and proceed with modeling the dependencies outwards.

- modeling of abstract concepts
- we do not design tables yet – we are focusing on the concept not the structure of the database
- the model has to ascertain that both the domain expert and the database designer fully understand each other, hence both of the have to understand the complete model
- designing the entities, determining their tasks and preparing them to be described in more detail on later stages

Main phases

- we make the conceptual model more technical
- it will serve as a base for implementation
- we eliminate the redundant entities, determine the relations between the entities

Structure diagram

We identify the main entities, the properties (attributes) of the entities, and the relationships between the entities. This information can be represented graphically:

- Chen's diagram
- IDEF1X
- Bachman's diagram
- Martin's diagram
- UML

Physical model of the data

- define the database components: tables, their columns, data types, keys (including foreign keys), constraints on column values, triggers (consistency checks), access levels, views and stored procedures
- we normalize the database (each table separately)
- we take into account the system constraints (e.g. dialect, operating system, frontend API) and the organizational constraints (e.g. expected number of users, number of user roles, event logging requirements, naming convention, maintenance schedule)

Moving from logical model to the physical

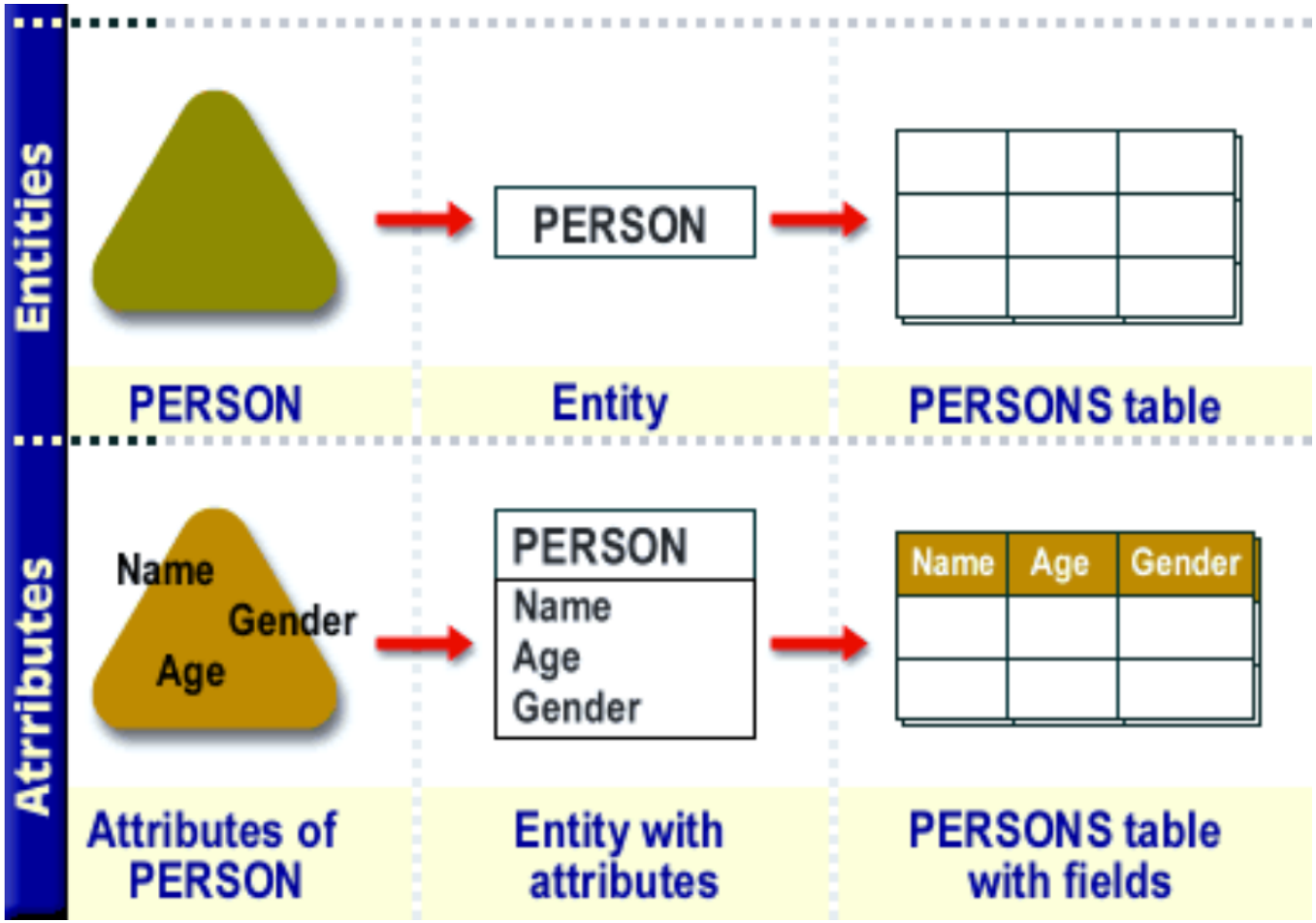
Depending on the type of database (relational, relational-object, object-oriented), difficulties may arise:

- implementing one-to-many and many-to-many relations
- lack of inheritance
- severely limited set of the datatypes
- limiting the types to atomic ones – some information require multiple records
- selecting a primary key (note the multiple records per 'single information')
- user-imposed constraints – execution time, memory usage, number of I/O operations (we may need denormalization)

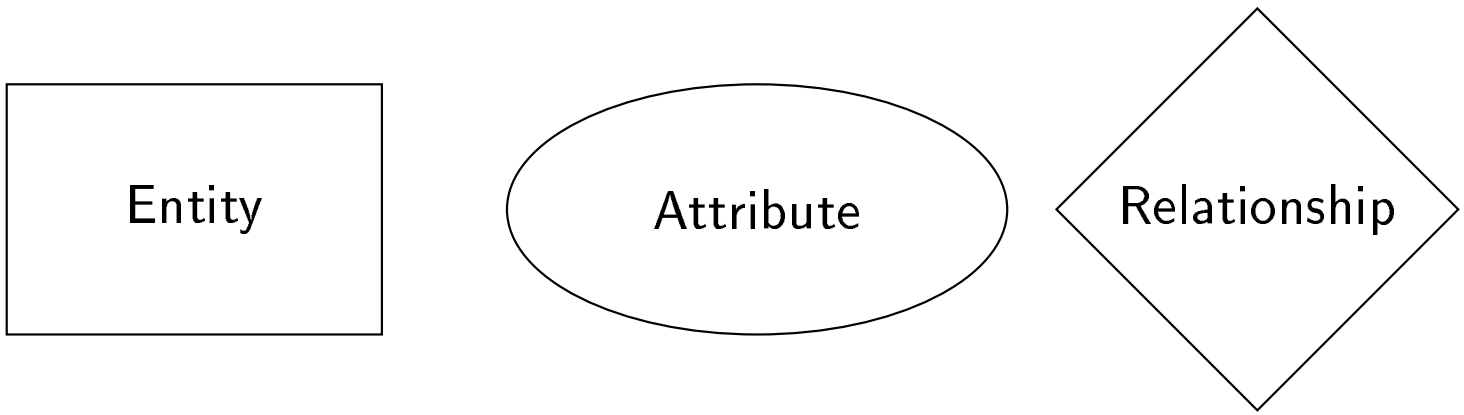
Summary

1. We determine the purpose of the database
2. We consult the domain-specific knowledge and restrictions
3. We determine the basic components of the database and identify where the bulk of the information lies, we determine the flow of the information
4. We model the entities, their attributes and the relations between the entities
5. We transform the entities (and some of the relations) into tables, we assign keys, triggers and constraints
6. We take into account the system and user-imposed constraints
7. We normalize the database
8. Testing and maintnance

Summary

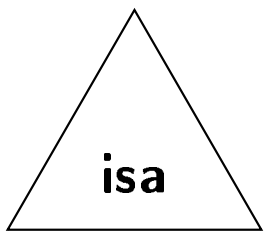

Entities
PERSON
PERSON
Entity
PERSONS table
Atrributes
Name
Gender
Age
PERSON
Name
Age
Gender
Name
Age
Gender
Attributes of PERSON
Entity with attributes
PERSONS table with fields

Basic elements



Entities

Entity is a specified concept during logical modeling, it's characterized by a regular structure of properties and relationships.
There may be a special type of relationship between entities – specialization



Attributes

- specify the properties of the entities
- specify the features or types of a relationship
- define the characteristics of a complex attribute
- we specify a subset of attributes that identify an entity (a primary key)

Relationships

They define the type of connection or association between the entities

- unambiguous relationships (1-1)
- functional relationships: one-to-many (1-n)
- multivariate associations: many-to-many (n-m)

Note that between the entities there may be some recursive links (autorelationships), e.g. between *person* entity for emloyer and emplyee. There might be also relationships between more than 2 entities.

Basic elements

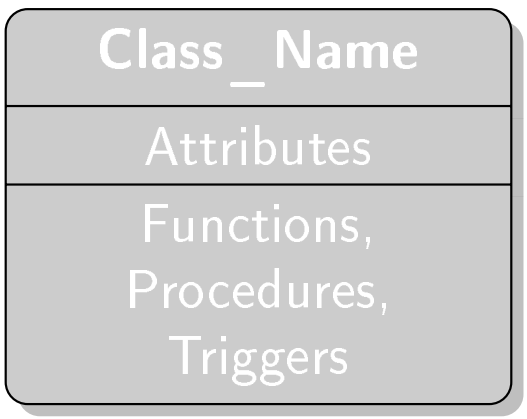
- enhances the basic Chen's notation (e.g. adding the activities)
- allows us to model events (flowcharts)
- extends the multiplicity modeling (optional entities)

Multiplicities:



Class diagrams

- each entity is represented by a separate class
- we may specify the entity at different level of details (class names only, adding attributes, adding functions and triggers, . . .)
- we can create class trees (inheritance) to allow different levels of entity detail



Associations

- indicate class relationships
- association label indicates multiplicity and direction of association
- in the case of autoassociation, it is necessary to specify the roles and the multiplicity at each end of the association
- an association can have its own attributes (association classes, connected by a dashed line to an association)

Aggregation and composition

- stronger types of relationships (required part of a whole)
- as opposed to association, the existence of one entity can be conditioned by the existence of another
- we denote the attributes and multiplicities as in association
- we denote them in the diagram indicated by an empty and a filled diamond, respectively

Association modeling

- one-to-one: we extend the table or use one of the solutions below
- one-to-many: we implement a table for each entity and implement foreign keys (or use the solution for many-to-many relationship)
- many-to-many: we implement a separate table (dictionary) for the relationship – we create foreign keys to both connected tables

Specialization and inheritance

In relational databases, the inheritance typical of object-oriented databases and programming languages has to be modeled in a different way:

- A single table for the entire class tree with a details attribute that marks the specialization – mandatory comments
- Adding the attributes of the main class and the inheriting class, we create separate tables for all derived classes
- Separating each class, modeling inheritance relationships by aggregation/composition with fixed multiplicities

Modeling NoSQL databases

- Each document may have a different structure
- No fixed relationships between collections
- We model the base with the application in mind (its restrictions and possibilities)
- Documents are modeled as entities aggregated with collection entities
- We model the connections between entities taking into account programming language classes/interfaces

Requirements examples

Description: Design a database for a football league.

Requirements examples

Description: Design a database for a football league. Include players, teams, and games played.

Requirements examples

Description: Design a database for a football league. There is always at least one team in the league, each team consists of multiple players and at least one coach. Teams are identified by their name and their headquarters, whereas people are identified by their first name, last name and date of birth. Additionally, information about goals, assists and cards for each player is stored. (For each team, on the other hand, we know the . . . .)
Each match takes place in a specific stadium, within a set competition. It is played by the home team and the visiting team, an ordered pair of teams along with the season of the game uniquely identifies the game. No team can play a match two days in a row. For each match we store its score, goal scorers, information about assists, cards, the referee of the match and additional statistics such as percentage of ball possession (the sum of ball possession by both teams can not exceed 100). The database should automatically update the players' statistics based on the added match. It should be possible to generate the current standings. After all matches have been played (a match and a rematch between each pair of teams) the season should be automatically ended.

Tool examples for ERD and UML

- GitMind Online
- Dia
- NetBeans UML
- Oracle J/SQL Developer
- Eclipse UML
- UML Designer
- Ubrello
- UMLet
- Papyrus
- UML Modeler
- Vertabelo
- TikZ – example, TikZ-UML package