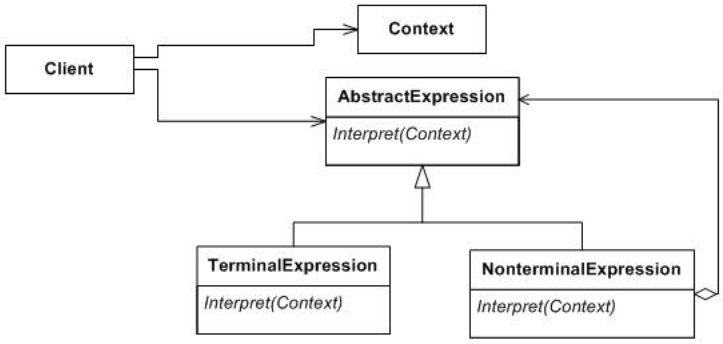
# Data Models

*In this document[1] I briefly overcome some concepts from software engineering that are needed to fully understand RDF and RDFS. Be sure to understand this document and ask the lecturers if you have any further question.*

A data model is a representation of real-world entities in a certain form. Computers need models to represent abstract and complex objects in a such a way our programs can manipulate them. One can say that data models sit at three different levels of abstraction:

- Conceptual data models,
- Logical data models,
- Physical data models.

A **conceptual model** is typically a diagrammatic representation of *objects* and *relationships* with no implementation details. For example:




The most popular conceptual modeling languages are the UML class diagram (http://www.uml.org/) and the ER language, which provide a diagrammatic representation of a *domain* (i.e., the knowledge we want to represent in our program; e.g., a program for a retailer company will talk about products, customers, shops, etc., which will form its domain).

Typically, in software engineering, software patterns (http://en.wikipedia.org/wiki/Software_design_pattern), such as the popular model view controller (MVC), are represented at the conceptual level.

Two main constructs appear at the conceptual level, objects (or classes, or elements, etc.) and relationships (or properties, etc.). For relationships, we typically distinguish certain subtypes, being the *association* (a relationship between two classes) and *subclass / superclass* (also called generalization / specialization; which depict taxonomies of concept or relationships) the most popular. Other popular types would be the aggregation and whole-part relationships.

[1] This document does not pretend to be a replacement of the software engineering courses, but just a quick reminder to refresh basic concepts that might not be fresh.

The conceptual design is typically carried out at the beginning of a project and it usually is the result of the requirement engineering phase, in which we try to understand what we need to build.

A **logical model** is a conceptual model represented according to a certain technology (or logical data model language). Typical logical data models are the relational model, XML, JSON, RDF, etc.

Bear in mind though that many people do not create conceptual models and directly deploy logical or physical models. The difference with a conceptual data model is that these ones tell us how to structure knowledge (the previous ones do not tell us how to represent the elements, they just depict them, i.e., focus on the what).

Those logical models used for representing data in a database are also called database models, and for a specific database model we usually refer to it as the database *schema*.

For example, given a UML class diagram (or conceptual model) we can model it as a relational schema by following some well-known design rules. This way, each conceptual class / object must be represented as a (logical) table whose identifier is asserted as the table primary key. Furthermore, conceptual relationships, according to the cardinality they hold, must be represented either as (logical) attributes + foreign keys, or as an associative class (for many-to-many relationships). Look, a (logical) associative table does typically not have a corresponding conceptual class (it maps to a relationship) and it is just the way the relational model has to deal with many-to-many relationships.

In general, a conceptual model is richer than a logical model (since we are not tied to a certain technology) and from the same conceptual model we may obtain many different logical models (or all of them valid).

At this stage, we normally start dealing with *instances*. Instances are single occurrences, typically elements created by the class Factory method in the object-oriented paradigm). Instances can also be represented at the conceptual level, but since conceptual modeling is created when still dealing at a higher abstraction level, many people do not represent them at all there.

In database models, we talk about the schema (or the *intentional* part, which is a representation of the classes and relationships; i.e., tables, rows and columns plus constraints such as the PK or the FK) and its instances (or the *extensional* part, which are the ocurrences inserted in the tables). This concept though, is not specific for databases and, in general, the schema (or model) tell us **how** to structure the instances.

Typically, logical models are created during the design phase and are typically intertwined with the implementation (and the design of the physical model).

I will not go further into **physical data models** but these are those already implemented in a software. For example, Oracle or PostgreSQL are two databases using the relational model but the internal structures they use to store tables, indexes, etc. are different and thus, yielding different physical schemas / models. Another example is a framework implementing a certain software pattern (e.g., AngularJS implements an evolution of the model-view-controller). This *pattern* is not any more at the conceptual level (indeed, it is not a pattern anymore, since it is been instantiated) and thus we say that AngularJS is a physical implementation of the MVC conceptual data model.

## RDF, RDFS and their map to data modeling

RDF and RDFS, as we saw them, sit at the logical level. They can be used to represent both schema and instances, which is the challenge of *semantic models*. The objective is to have a language allowing us to represent both the schema (in a rich way) and the instances so that we can publish or open data and someone who had never seen that dataset can exploit the instances (since by checking the schema will be able to both understand and know how the instances are structured in such dataset).

Thus, in RDF and RDFS we can describe the schema and, for example, talk about the customer, shop, buy and revenue concepts. All of them are schema concepts, and we can create relationships between them (subClassOf, subPropertyOf, domain, range, etc. which are the specific notation used by the RDF technology). In a traditional software design setting, this kind of relationships are those you first think of at the conceptual level.

Also, RDF can also represent instances such as customer1, shop1, etc. and represent relationships between these instances by using the schema properties (e.g., buys). In a traditional software engineering setting, this is the kind of task we would start thinking of at the logical /physical level (i.e., create and manage instances).

Finally, and importantly, RDF allows us to *type* a certain instance with its class. Note this is the badly needed link between schema and instances. We can say that customer1 is an instance of customer (by using rdf:type) and shop1 is an instance of shop (by using, again, rdf:type).

What you need to learn from this session is the meaning of the main RDF and RDF constructs (namely, rdf:type and the rdfs classes and properties introduced in the first lecture) in order to be able to express in these languages any domain. Once this is clear, querying RDF and RDFS with SPARQL will be much easier.