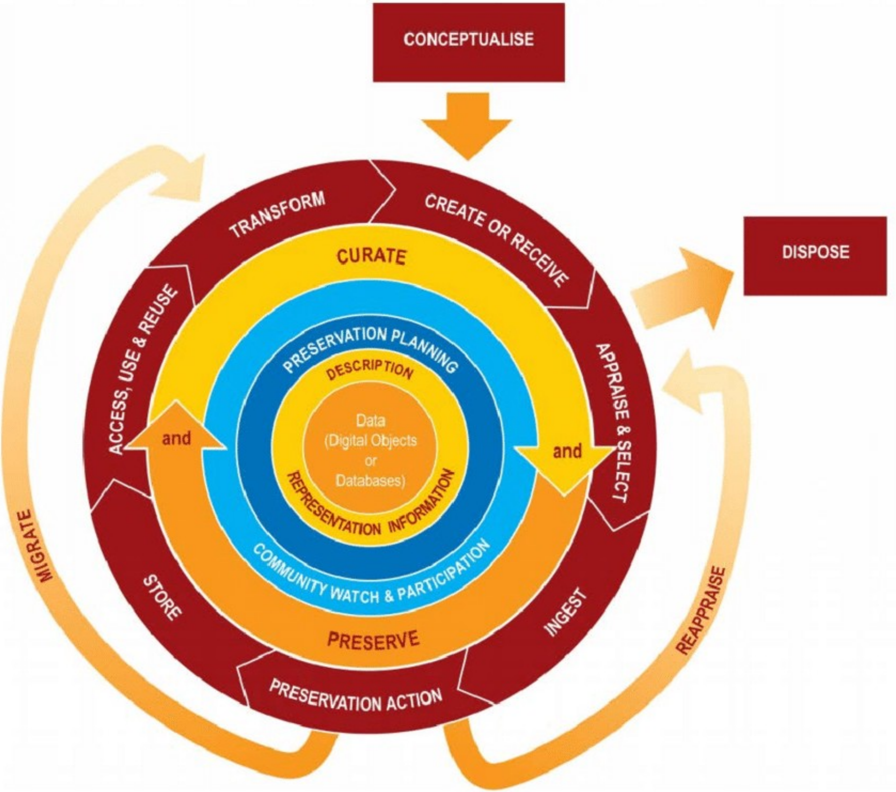
Data Model

Andrea Bignamini - INAF-OATs

Data Life Cycle – Digital Curation Centre



Source:
https://www.dcc.ac.uk/guidance/curation-lifecycle-model

Scientific data management

- **During the scientific exploration process, from the data generation phase to the data analysis phase, data management involves several aspects**
  - the efficient access to storage systems, in particular, parallel file systems, to write and read large volumes of data
  - the efficient data movement and management of storage spaces
  - techniques for automatically optimizing the physical organization of data, necessary for fast analysis
  - techniques to effectively perform complex data analysis and searches over large datasets
  - the automation of multistep scientific process workflows

The FAIR Guiding Principles

To be Findable:

F1. (meta)data are assigned a globally unique and persistent identifier
F2. data are described with rich metadata
F3. metadata clearly and explicitly include the identifier of the data it describes
F4. (meta)data are registered or indexed in a searchable resource

To be Accessible:

A1. (meta)data are retrievable by their identifier using a standardized communications protocol
A1.1 the protocol is open, free, and universally implementable
A1.2 the protocol allows for an authentication and authorization procedure, where necessary
A2. metadata are accessible, even when the data are no longer available

To be Interoperable:

I1. (meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation
I2. (meta)data use vocabularies that follow FAIR principles
I3. (meta)data include qualified references to other (meta)data

To be Reusable:

R1. meta(data) are richly described with a plurality of accurate and relevant attributes
R1.1. (meta)data are released with a clear and accessible data usage license
R1.2. (meta)data are associated with detailed provenance
R1.3. (meta)data meet domain-relevant community standards

My Datasets...

Storage and Archive

- Data structure, consistency, cleanness, order, organization are key point to reach the goal of find useful information in millions or billions of digital objects.
- Distinction between collections, organization of different sized datasets, timing of data usage are fundamental considerations to organize your data

Metadata

- Metadata are data that describe other data
- For example *author*, *date created* and *date modified* and *file size* are very basic document metadata
- Metadata are data themselves
- Metadata are essential for:
  - Data description
  - Data discovery
  - Data linking
- Metadata must store all information necessary to understand and use data

Data Management & Stewardship

- **Good Data Management and Stewardship is the key that leads to:**
  - Knowledge discovery and innovation
  - Data and knowledge integration and reuse by the community
- **The problem is far beyond long term data storage, since it includes data annotation** → Metadata!

Goal:

- **transparency**
- **reproducibility**
- **reusability**

of data holdings for

- **humans**
- **machines**

Human-driven activities

- **Intuitive sense of semantics**
  - Ability to identify directly the context(s)
- **Less prone to error in selecting the data**
  - Caveat: also humans need metadata
- **Not fit to scope, scale, speed**
  - Big Data → We need machines!

Machine-driven activities

- **Must be able to face wide range of**
  - Types
  - Formats
  - Protocols
- **Must keep provenance records**

"Machine Actionability"

- **Requires datasets with detailed information to move through autonomous action steps**
  - Identify object type
  - Determine usefulness interrogating metadata
  - Determine usability: license, accessibility…
  - Take appropriate action

Define your Use Cases

- What data will you collect or create?
- How will the data be collected or created?
- What metadata will describe the data?
- Do the datasets I'm searching for already exist?
- What tools I use?
- What formats are available?
- Can my data be used together with other dataset from different repositories?
- Who are the end users or the reference community who will use this data?
- How do I access them?
- How will you share the data?
- Who will be responsible for data management?
- Can all of this be automated or does it require human intervention?

Data formats

- **FITS is the standard data format used in Astronomy**
  - ESA and NASA developed FITS in the late 1970s, stemming from radio astronomy (FITS is always backward compatible)
  - The Vatican Library has adopted the FITS data format for the long-term digital preservation of the books, manuscripts, and other objects in its vast collection
- **HDF5**
  - used in several research areas, including earth sciences, computational fluid dynamics, astronomy, astrophysics, but also financial services and industry
- **NetCDF is a set of interfaces for array-oriented data access. Starting with version 4, the netCDF library can use HDF5 files as its base format**
  - Used in climatology, meteorology and oceanography applications (e.g., weather forecasting, climate change) and GIS applications
- **ROOT**
  - Originally designed for particle physics (at CERN), its usage has extended to other data-intensive fields like astrophysics and neuroscience

File formats features

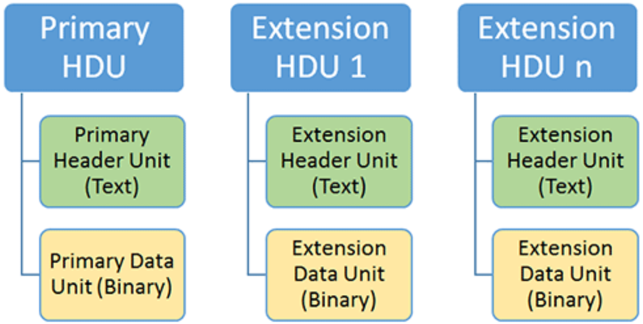
- **Self-description (i.e. metadata)**
  - Human-readable metadata availability
- **Open-format, i.e. with a public specification maintained by a standards organization**
- **Machine independence**
- **Storage efficiency**
- **Data structures: images, n-dimensional arrays, tables, objects sequences, hierarchical structures**
- **Internal data compression (e.g. tile compression)**
- **Data access**
  - read/write a portion of the n-dimensional arrays (hyperslabs) or tables

FITS format

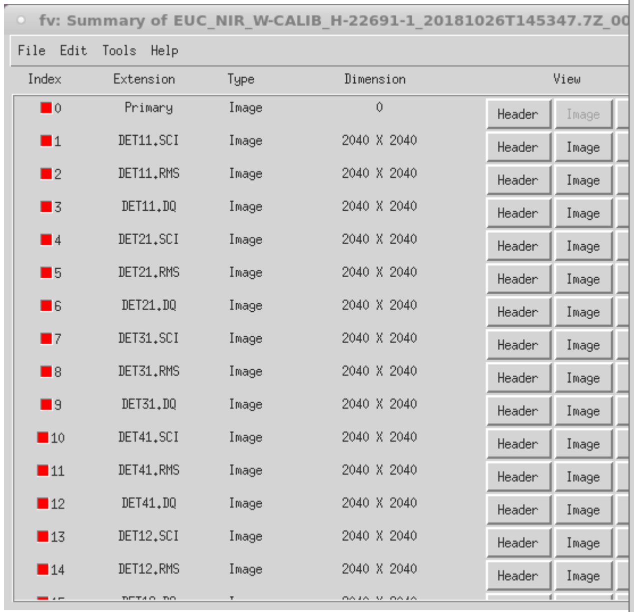
- **Even if mainly used in Astronomy, it is useful to start with a quick view of the FITS standard, in order to highlight some concepts and data structures**
- **The first FITS (Flexible Image Transport System) standard was published in 1981. The most recent version (4.0) has been standardized in 2016**
  - Ref: https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf
- **It is primarily designed to store scientific data sets consisting of multidimensional arrays (images) and 2-dimensional tables organized into rows and columns of information**
- **In few words a FITS file is composed by two distinct parts, which can be repeated several times:**
  - the first part (**header**) is formed by easily viewable ASCII text elements providing metadata information
  - in the second part there are the data in **binary format** (a multi-dimensional array or a table)

The FITS HDU

- **The header and the binary part together are called Header Data Unit (HDU)**
  - The binary part (data unit) is always optional
  - The first HDU is called **primary HDU** or primary array and its binary part can only be an image (n-dimensional array)
  - Any number of additional HDUs may follow the primary array. These additional HDUs are referred to as FITS ‘extensions’
  - The binary part of a fits extension can contain either an n-dimensional array or a table
    - To be precise, the data unit can also contain an ASCII table, so it is not always binary



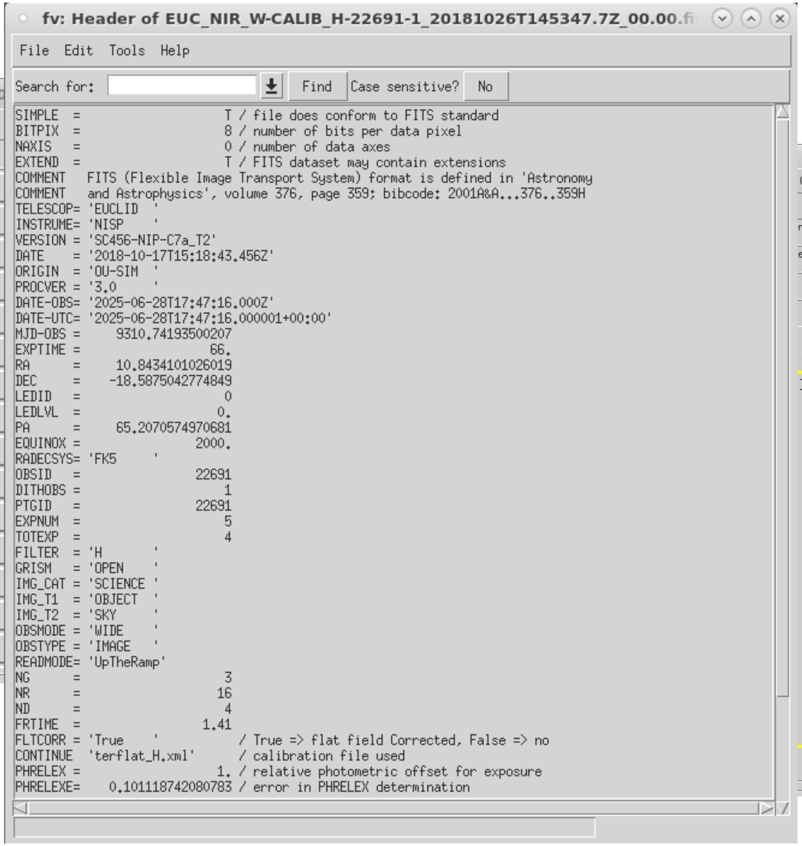
FITS header example: NISP example



fv: Summary of EUC_NIR_W-CALIB_H-22691-1_20181026T145347.7Z_00

File Edit Tools Help

Index	Extension	Type	Dimension	View	
0	Primary	Image	0	Header	Image
1	DET11.SCI	Image	2040 X 2040	Header	Image
2	DET11.RMS	Image	2040 X 2040	Header	Image
3	DET11.DQ	Image	2040 X 2040	Header	Image
4	DET21.SCI	Image	2040 X 2040	Header	Image
5	DET21.RMS	Image	2040 X 2040	Header	Image
6	DET21.DQ	Image	2040 X 2040	Header	Image
7	DET31.SCI	Image	2040 X 2040	Header	Image
8	DET31.RMS	Image	2040 X 2040	Header	Image
9	DET31.DQ	Image	2040 X 2040	Header	Image
10	DET41.SCI	Image	2040 X 2040	Header	Image
11	DET41.RMS	Image	2040 X 2040	Header	Image
12	DET41.DQ	Image	2040 X 2040	Header	Image
13	DET12.SCI	Image	2040 X 2040	Header	Image
14	DET12.RMS	Image	2040 X 2040	Header	Image

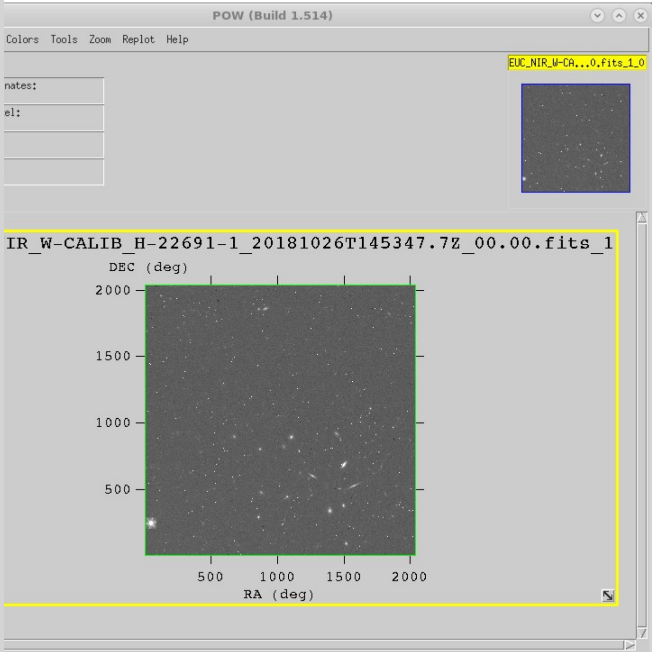


fv: Header of EUC_NIR_W-CALIB_H-22691-1_20181026T145347.7Z_00.00.fi

File Edit Tools Help

Search for: Find Case sensitive? No

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                    8 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
EXTEND  =                    T / FITS dataset may contain extensions
COMMENT   FITS (Flexible Image Transport System) format is defined in 'Astronomy
COMMENT   and Astrophysics', volume 376, page 359; bibcode: 2001A&A...376..359H
TELESCOP= 'EUCLID  '
INSTRUME= 'NISP    '
VERSION = 'SC456-NIP-C7a_T2'
DATE    = '2018-10-17T15:18:43.456Z'
ORIGIN  = 'OU-SIM  '
PROCVER = '3.0     '
DATE-OBS= '2025-06-28T17:47:16.000Z'
DATE-UTC= '2025-06-28T17:47:16.000001+00:00'
MJD-OBS =     9310.74193500207
EXPTIME =                  66.
RA      =     10.8434101026019
DEC     =    -18.5875042774849
LEDID   =                    0
LEDLVL  =                   0.
PA      =     65.2070574970681
EQUINOX =                2000.
RADECSYS= 'FK5     '
OBSID   =                22691
DITHOBS =                    1
PTGID   =                22691
EXPNUM  =                    5
TOTEXP  =                    4
FILTER  = 'H       '
GRISM   = 'OPEN    '
IMG_CAT = 'SCIENCE '
IMG_T1  = 'OBJECT  '
IMG_T2  = 'SKY     '
OBSMODE = 'WIDE    '
OBSTYPE = 'IMAGE   '
READMODE= 'UpTheRamp'
NG      =                    3
NR      =                   16
ND      =                    4
FRTIME  =                 1.41
FLTCORR = 'True    '           / True => flat field Corrected, False => no
CONTINUE  'terflat_H.xml'      / calibration file used
PHRELEX =                   1. / relative photometric offset for exposure
PHRELEXE=    0.101118742080783 / error in PHRELEX determination
```



Credit to Marco Frailis

FITS metadata and data

- **FITS keywords are defined by a keyword name, a value (string, logical, int, float, complex) and an optional comment**
  - The comment is used to further document the metadata information, e.g. indicating the unit of measure and purpose or, for date time values, the epoch used
  - Keyword names are limited to 8 characters, but a widely used standard extension allows longer names
- **The FITS standard also fixes a dictionary of keyword names and corresponding value type and format for representation of World Coordinate Systems and time coordinates**
- **Additional dictionaries are defined by astronomy organizations such as the European Southern Observatory (ESO) and the National Optical Astronomy Observatory (NOAO)**

FITS Keyword Dictionaries

The following data dictionaries contain compilations of the FITS header keywords that have been defined and used within various contexts.

- Keywords defined in the FITS Standard
- Other commonly used keywords
- UCO/Lick keyword dictionary
- STScI keyword dictionary
- NOAO keyword dictionary
- ESO keyword dictionary

Metadata and provenance

- **Today, the key drivers for the capture and management of data descriptions are the scientific collaborations**
  - They bring collective knowledge and resources to explore a research area
- **These data need to contain enough information so that members of the collaboration can interpret them and use them for their research**
- **Metadata and provenance information are also important for the automation of scientific analysis**
  - Analysis software needs to
    - be able to identify the datasets appropriate for a particular analysis
    - annotate new, derived data with metadata and provenance information

I put a lot of metadata, but my data are still a mess! How can I add value to them?

JSON

- JSON (JavaScript Object Notation) is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition
- JSON is a way of storing and communicating data with specific rules (like XML, YAML, etc.)
- JSON files has extension .json
- JSON uses key-value pairs
- JSON was designed to be human and machine readable
- JSON is easy to read and write
- Language independent even if it comes from JavaScript

https://www.json.org

JSON Example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "streetAddress": "21 2nd Street",
  "city": "New York",
  "state": "NY",
  "postalCode": "10021-3100",
  "homePhoneNumber": "212 555-1234",
  "officePhoneNumber": "646 555-4567",
  "child1": "Catherine",
  "child2": "Thomas",
  "child3": "Trevor",
  "spouse": null
}
```

Source:
https://en.wikipedia.org/wiki/JSON#Syntax

JSON Example

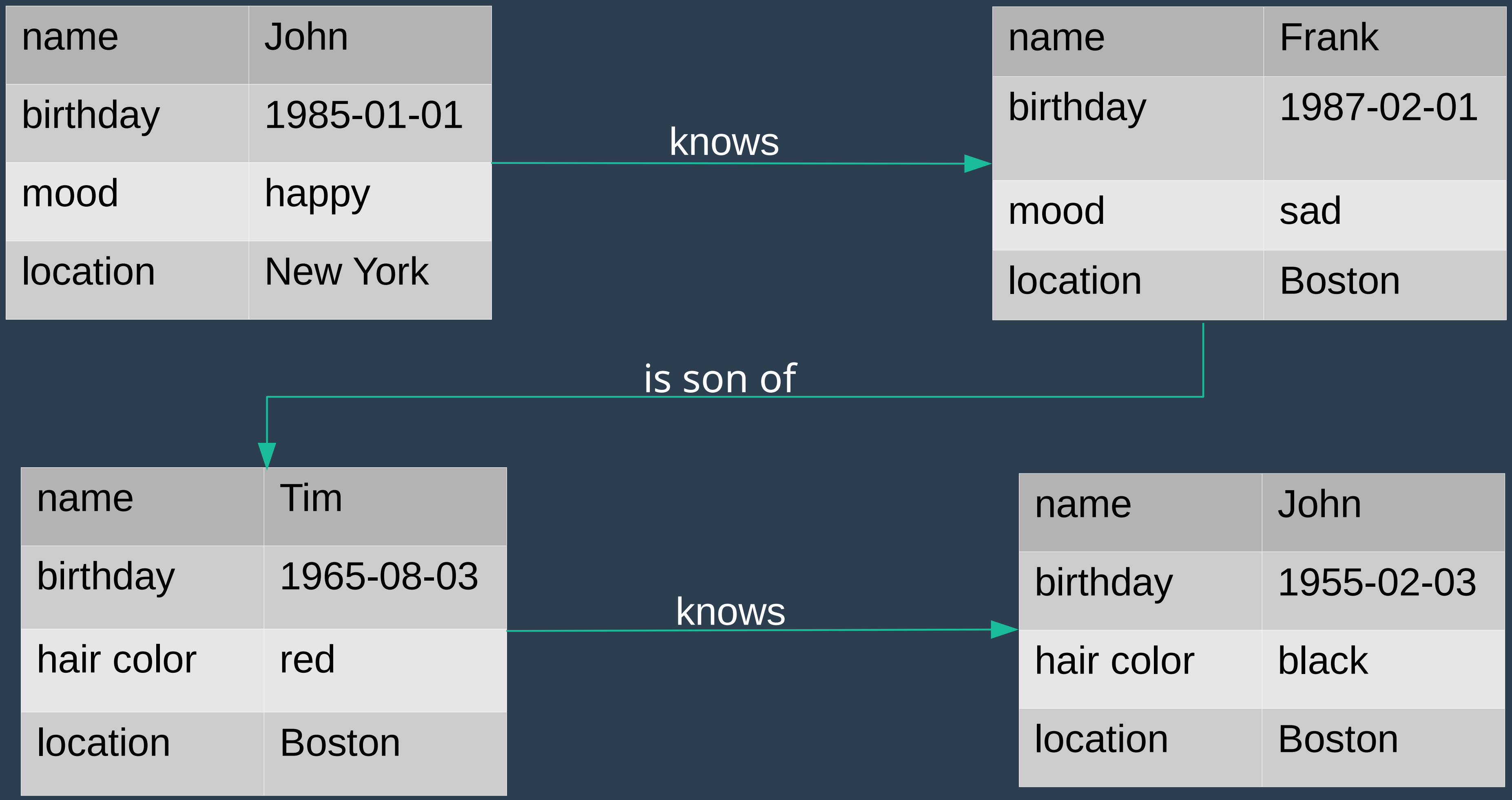
```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "streetAddress": "21 2nd Street",
  "city": "New York",
  "state": "NY",
  "postalCode": "10021-3100",
  "homePhoneNumber": "212 555-1234",
  "officePhoneNumber": "646 555-4567",
  "child1": "Catherine",
  "child2": "Thomas",
  "child3": "Trevor",
  "spouse": null
}
```

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
      "Catherine",
      "Thomas",
      "Trevor"
  ],
  "spouse": null
}
```

Source:
https://en.wikipedia.org/wiki/JSON#Syntax

name	John
birthday	1985-01-01
mood	happy
location	New York

- Data are in a standard and usable format
- Common formats for data:
  - Json
  - XML
  - CSV
  - RDF
- How “John” is related with the rest of the world?



- Relations link “John” with the rest of the world http://mysite.com/john

name	John
birthday	1985-01-01
mood	happy
location	New York

http://schema.org/knows

http://mysite.com/frank

name	Frank
birthday	1987-02-01
mood	sad
location	Boston

http://schema.org/isSonOf

http://othersite.com/tim

name	Tim
birthday	1965-08-03
hair color	red
location	Boston

http://schema.org/knows

http://othersite.com/john

name	John
birthday	1955-02-03
hair color	black
location	Boston

- URL can specify which “John”
- URL can define relations

http://mysite.com/john

name	John
birthday	1985-01-01
mood	happy
location	New York

http://schema.org/knows

http://mysite.com/frank

name	Frank
birthday	1987-02-01
mood	sad
location	Boston

http://schema.org/lives

http://towns.com/newyork

name	New York
latitude	40°43'N
longitude	74°00'W
state	New York

http://schema.org/lives

http://towns.com/boston

name	Boston
latitude	41°21'N
longitude	71°03'W
state	Massachusetts

- You can mix information from different vocabularies

John: “Do you have any questions?”

Why data modeling

- **Quality:** conceptual integrity is the most important consideration in system design
- **Communication:** models reduce misunderstandings and promote consensus among developers, customers, and other stakeholders
- **Reliability:** rigorous modeling improves the quality of the data. You can weave constraints into the fabric of a model and the resulting database
- **Performance:** a sound model simplifies database tuning

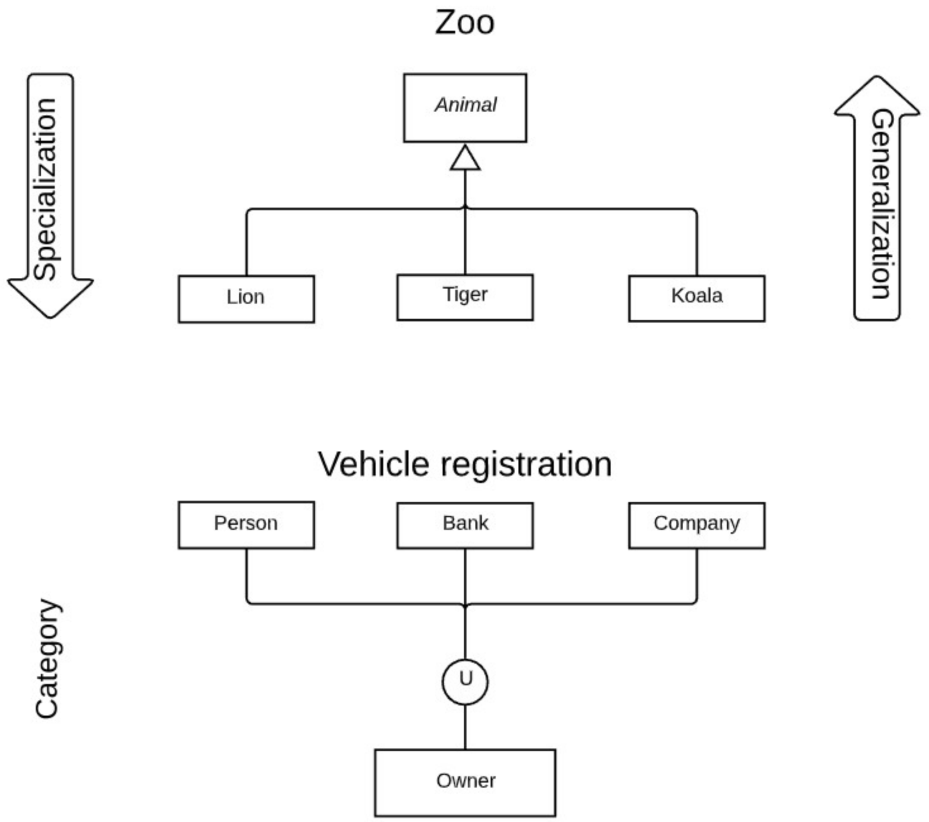
Data model (1)

- **A model is a representation of some aspect of a problem that lets you thoroughly understand it**
- **A data model is a model that describes how data is stored and accessed**
  - It does not include many of the details of how the data is stored or how the operations are implemented
  - It uses logical concepts, such as objects, their properties, and their interrelationships
- **Categories of data model**
  - **Conceptual data model**: focuses on major entity types and relationship types. Provides a high-level overview. Has no attributes
  - **Logical data model**: fleshes out the conceptual model with attributes and lesser entity types
  - **Physical data model**: converts the logical model into a database design. The emphasis is on physical constructs such as tables, keys, indexes, and constraints.

Data model (2)

- **Entity-Relationship (ER) models**
  - **Entity**: real-world object or concept
  - **Attribute**: property of interest that further describes the entity
  - **Relationship**: among two or more entities, it represents the associations among the entities
- **Additional abstractions for advanced ER models:**
  - **Specialization**: Specialization is the process of defining a set of subclasses of an entity type; this entity type is called the superclass of the specialization
    - The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass
  - **Generalization**: reverse process of abstraction in which we identify the common features of several entities, and generalize them into a single superclass of which the original entity types are special subclasses
  - **Categories**: to represent a collection of entities from different entity types

Specialization, Generalization, Category



Data Modeling Methodologies

- **The process of designing a data model involves producing the previously described three types of schemas: conceptual, logical, and physical**
- **The approach will depends on your datasets, use cases, and requirements**
- **A fully attributed data model contains detailed attributes and relationships for every entity within it**
- **There are two may modeling approach:**
  - **Bottom-up**: you may start usually with existing data structures or databases to derive the physical data model
  - **Top-down**: you start in an abstact way from the conceptual data model adding details bit by bit
- **Do not reinvent the wheel!**
  - Standard data models for your datasets may already exist → Search for them!
  - Adopting standards will increase the Interoperability level of your project

The Zen of Python – PEP 20

```
>>> import this

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Entity-relationship diagrams

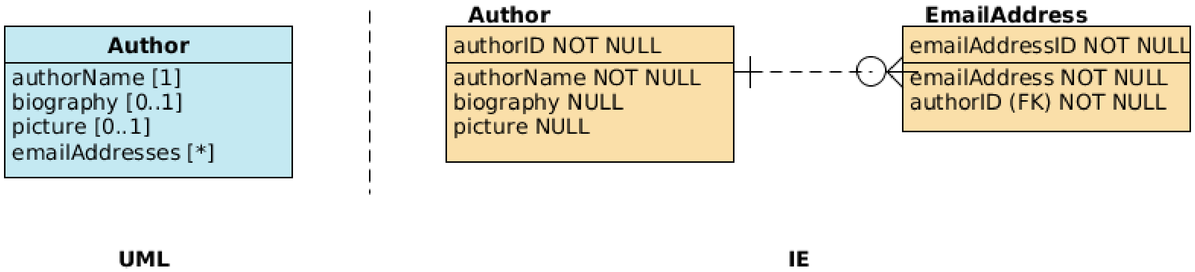
- Many different Entity-Relationship Diagrams (ERD) notations available
  - Chen notation, **Information Engineering** (**IE**) or Crows's foot notation, IDEF1X, **Unified Modeling Language (UML)**, etc.
- The Information Engineering is a modeling notation that has been in use for many years
- IE focuses on details such as tables, keys, and indexes (it is closer to the Physical data model). IE's attention to database detail is helpful for explaining nuances of the UML
- The IE lacks a standard notation and there are several variants
- The UML class model specifies classes (entity types) and their relationship types. It is closer to the Conceptual data model:
  - More concise than traditional database notations (usually no keys, foreign keys, indexes and referential integrity)
  - It provides an higher level of abstraction

UML class diagram

- **The UML class diagram specifies classes (entity types) and their relationship types**
- **An *object* is a concept, abstraction, or thing that has identity and meaning for an application**
  - Application needs also determine the level of abstraction for representing an object
  - E.g.: an airplane flight can be represented by departure/arrival time or as a sequence of phases (at gate, boarding, taking off, en route, landing, at gate, disembarking) depending on the applications
- **A class describes a group of objects with similar properties (attributes), behavior (operations), relationships to other objects, and semantic intent**

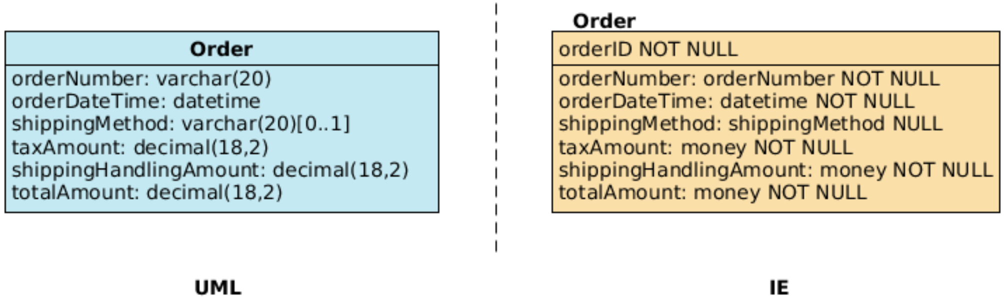
Classes and attributes

- **An attribute is a named property of a class that describes a value held by each object of the class**
  - The second portion of the UML class box shows attribute names
- **The IE notation lists attributes in both portions of the entity type box.**
  - The top portion has primary key attributes, the lower portion has the remaining data attributes
  - The attribute authorID above is a surrogate key (a generated number that uniquely identifies an author)
- **In UML, each attribute can have an attribute multiplicity that specifies the number of possible values for each record. If not specified, it defaults to [1].**
- **Normally, a relational database attribute cannot store a collection of values**
  - For IE, we had to convert the “many” multiplicity to a relationship type



Data types

- **It is good database practice for developers to assign each attribute a domain (IE) and then separately resolve the domain to a data type**
  - Flexibility: there are fewer domains than attributes
  - A domain can define both a data type and additional constraints
- **But most UML tools just assign each attribute a data type**
- **The UML notation lists the attribute name, a colon, the data type, and attribute multiplicity**
- **The IE notation lists the attribute name, a colon, the domain (optional), the data type (optional, can appear with or without the domain), and nullability**

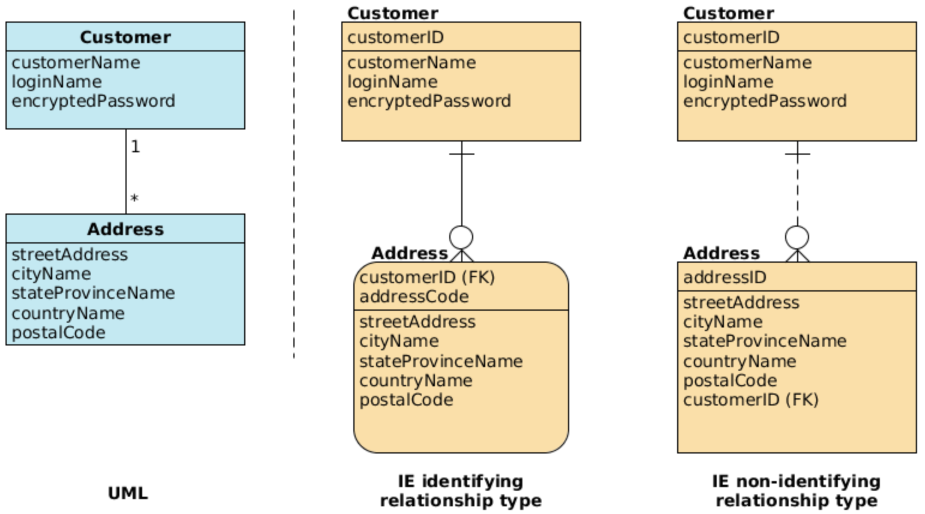


Notation summary (1)

UML Concept	UML Notation	IE Concept	IE Notation	Definition
Object		Entity		A concept, abstraction, or thing that has identity and meaning for an application.
Class		Entity type		A group of objects with similar attributes, behavior, relationships to other objects, and semantic intent.
Value		Value		A piece of data that lacks identity.
Attribute		Attribute		A named property of a class that describes a value held by each object of the class.
Operation				A function or procedure that can be applied to or by objects in a class.
		Domain		The named set of possible values for an attribute,
Data type		Data type		A specification of type and size for values such as long integer, varchar(20), and date.

Associations and Relationship

- **Associations provide the means for relating classes**
- **The UML notation for an association is a line**
- **A UML association corresponds to an IE:**
  - **Identifying relationship type** (solid line) the existence of the child entity relies solely on the parent entity
  - **Non-identifying relationship type** (dashed line) the child entity can stand on its own without the parent entity



Independent and dependent entity types

- **IE distinguishes between independent entity types** (square box) **and dependent entity types** (rounded box)
  - An independent entity type (also called **strong entity type**) does not include any foreign keys in its primary key. The IE symbol is a square-corner box
  - A dependent entity type (also called **weak entity type**) includes one or more foreign keys in its primary key (via one or more identifying relationship types or via generalization, see the next slides). It can exist only if one ore more other entity types also exist. The IE symbol is a rounded-corner box
- **IE distinguishes between identifying relationship type** (solid line) and **non-identifying relationship type** (dashed line)
  - An identifying relationship type propagates primary key attributes of the source entity type to the primary key of the referent entity type. A solid line connects the entity types. The referent entity type is necessarily dependent (rounded box).
  - A non-identifying relationship type propagates primary key attributes of the source entity type to data attributes of the referent entity type. A dashed line connects the entity types. The referent entity type may be independent (square box) or dependent (rounded box) depending on its other relationship types and generalizations (next slides).

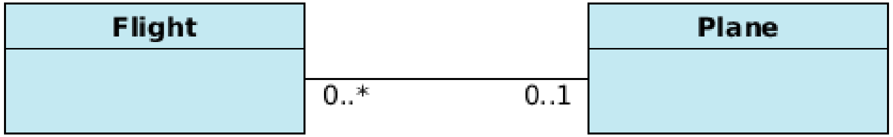
Multiplicity

- Multiplicity specifies the number of occurrences of one class that may relate to a single occurrence of an associated class
- Thus multiplicity pertains to an association end

Notation	Meaning
	Relationship
	One
	Many
	One and only one
	Zero or one
	One or many
	Zero or many

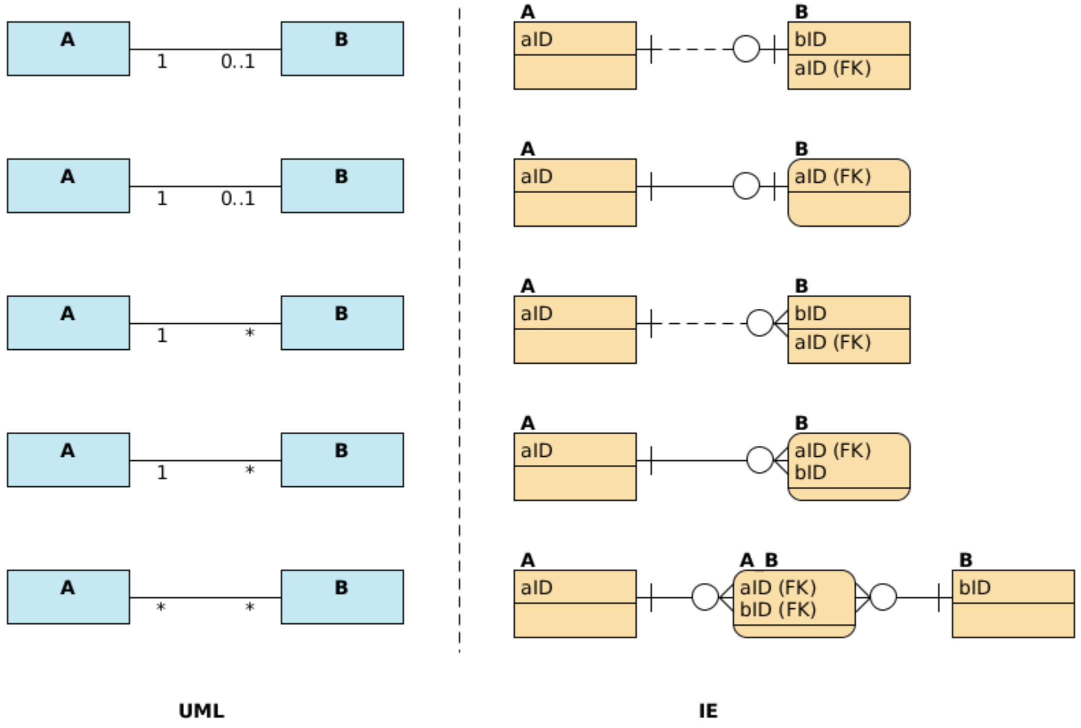
IE relationship symbols

UML multiplicity



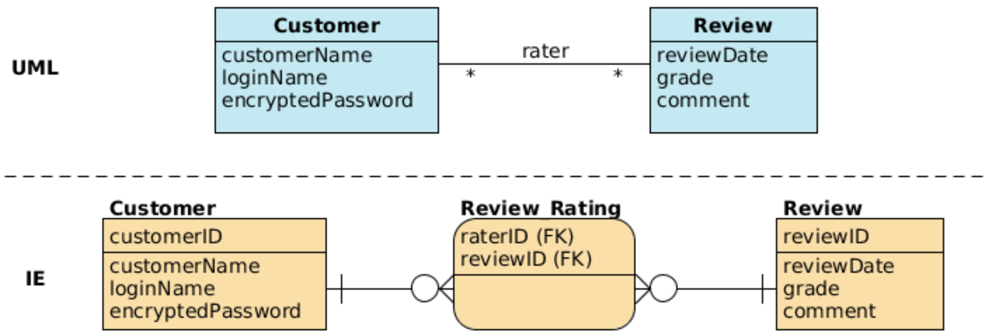
Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

Multiplicity: UML vs IE



Many-to-many relationships

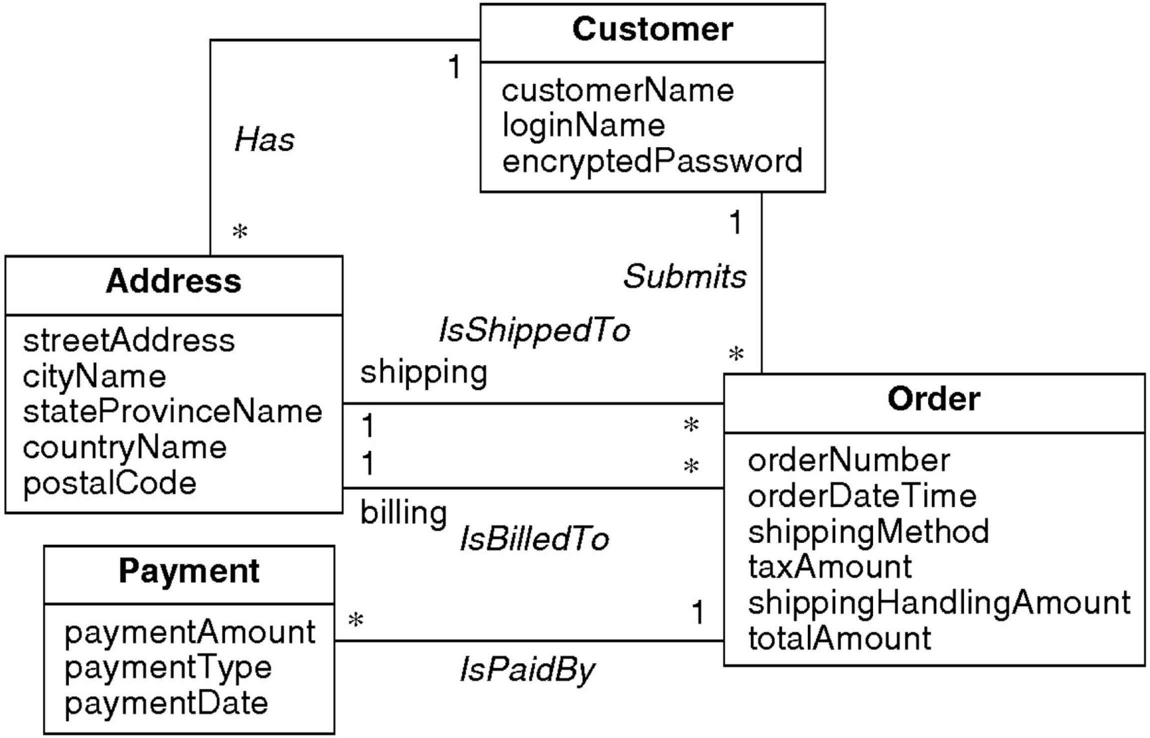
- **This example shows a many to many relationship between Customer and Review, i.e. a customer can rate 0 or more reviews and a review can be rated by 0 or more customers**
- **The physical data model of the IE notation shows that this type of relation is realized with a dependent entity type (Review_Rating) and two or more identifying relationship types**
- **Review_Rating is called an associative entity type**, i.e. it obtains its primary key from two or more entity types.
  - Review_Rating.raterID refers to Customer.customerID



Association names (UML)

- **The UML only requires association names when there are multiple associations between the same classes**
- **An association name often reads in a particular direction. Nevertheless, associations can be traversed either way**
  - The UML also has a navigation icon to show the direction for reading the name
- **This association traversal is analogous to combining relational database tables via foreign-key-to-primary-key joins**
- **An association end name is an alias for a class in an association. The UML notation is a legend next to the class-association intersection**
  - Association end names are optional if a model is unambiguous
  - Ambiguity occurs when there are multiple associations for the same classes or an association for objects of the same class
- **When constructing models, you should properly use association ends and not introduce a separate class for each reference**

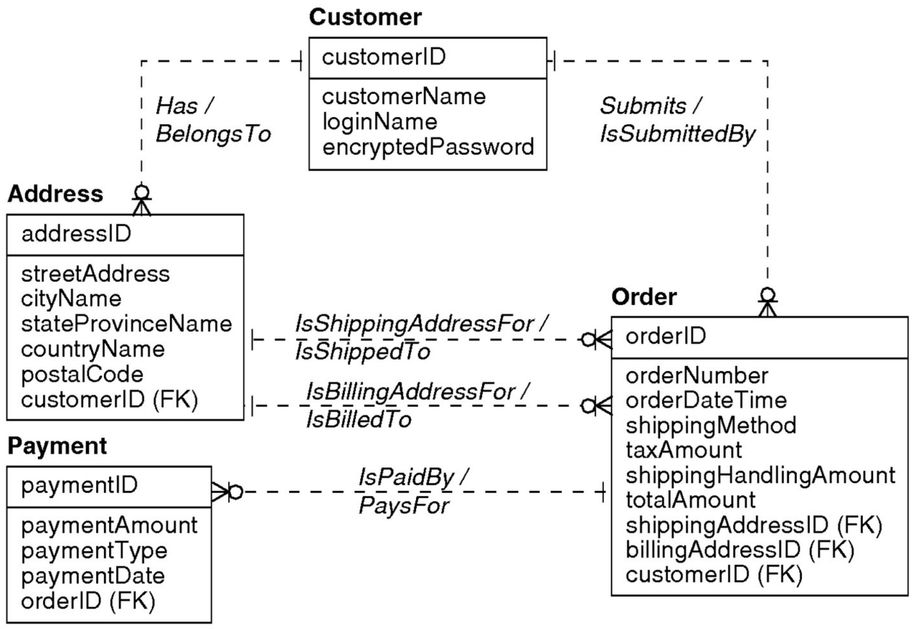
Association names example



Benefits of association names

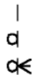
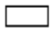
- **Benefits of association names:**
  - Improves model readability
  - Provides a table name for an associative entity type
  - Disambiguates multiple associations for the same classes
- **Benefits of association end names:**
  - Improves model readability
  - Provides a foreign key name
  - Disambiguates multiple associations for the same classes
  - Disambiguates an association for objects of the same class
  - Provides clarity for model traversal and SQL queries

Relationship names (IE)



- It is a common IE practice to include relationship type names. Each relationship type can have either a single name or a pair of directed names. Directed names add bulk but make a model more readable.
- A single name can be useful for development (it provides a table name)

Notation summary (2)

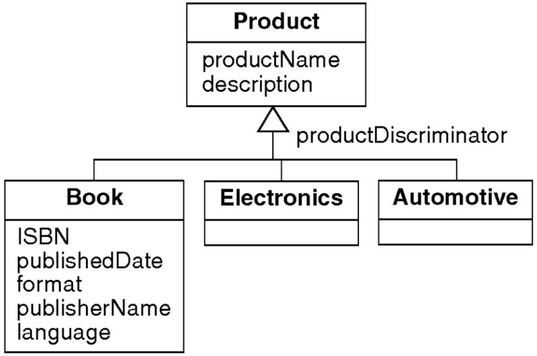
UML Concept	UML Notation	IE Concept	IE Notation	Definition
Link		Relationship		A physical or conceptual connection among objects.
Association	———	Relationship type	- - - - - ———	Describes a group of links with common structure and semantics.
Association end		Role		The use of a class in an association.
		Associative entity type		Resolves a many-to-many relationship type.
Multiplicity	1 0..1 *	Cardinality (though technically incorrect)		Specifies the number of occurrences of one class that may relate to a single occurrence of an associated class.
		Identifying relationship type	———	Propagates source primary key attributes to the referent primary key.
		Non-identifying relationship type	- - - - -	Propagates source primary key attributes to referent data attributes.
		Independent entity type		Has no foreign keys in its primary key. Also called a strong entity type.
		Dependent entity type		Includes foreign key(s) in its primary key. Also called a weak entity type.

Generalization

- **Generalization is a defining characteristic of object-oriented software approaches and organizes classes by their similarities and differences**
  - It leads to smaller models with deeper insight
- **Generalization couples a class (the superclass) to one or more variations of the class (the subclasses)**
- **The superclass holds common information (attributes, operations, and associations)**
- **Each subclass adds specific information**
- **Generalization organizes classes by their similarities and differences, structuring the description of objects. Generalization can arise from requirements that list structural alternatives**
- **The UML notation for generalization is a large hollow arrowhead that points to the superclass.**

Example of inheritance in UML

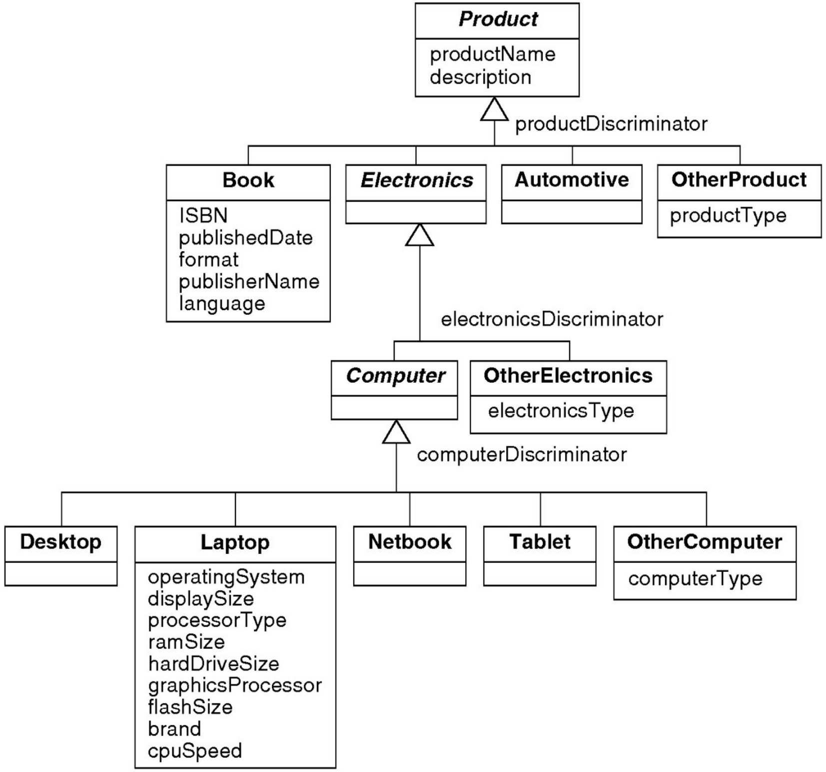
- **The generalization set name (productDiscriminator) is an enumerated attribute that can be placed next to the generalization symbol**
- **Generalization has two purposes**
  - Reuse. Subclasses can share information that superclasses provide
  - Form a taxonomy and declare what is similar and what is different about classes. This is much more profound than modeling each class individually and in isolation



Abstract class

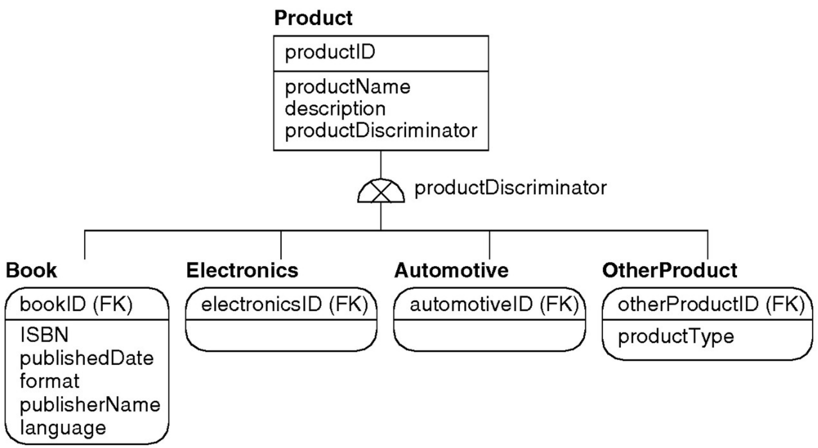
- An **abstract class** is a class that has no direct occurrences. The UML indicates an abstract class by italicizing the class name or placing the legend {abstract} before or after the class name
- A superclass can be abstract or concrete, depending on how the generalization is stated
- As a matter of style, it is a good idea to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance; all superclasses are abstract and all leaf subclasses are concrete.
- Deeply nested generalizations… try to avoid!

Nested generalization

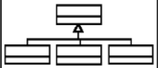
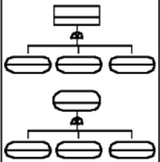
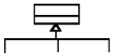
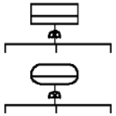
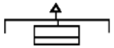
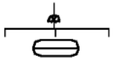


IE notation for generalization

- IE subtypes are dependent entity types because each subtype primary key refers to the supertype primary key
- The supertype may be independent or dependent (but is usually independent) based on whether its primary key incorporates a foreign key from another entity type.

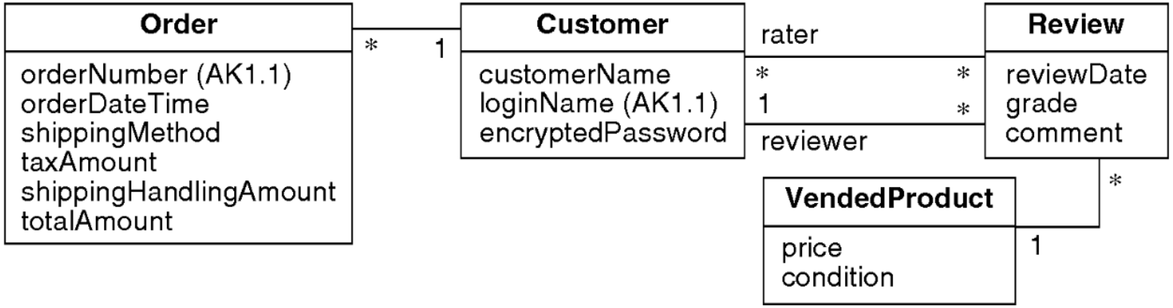


Notation summary (3)

UML Concept	UML Notation	IE Concept	IE Notation	Definition
Generalization		Subtyping		An organization of classes by their similarities and differences, structuring the description of objects.
Superclass		Supertype		The common attributes, operations, and associations for a generalization.
Subclass		Subtype		Specific attributes, operations, and associations for a generalization.
Generalization set name		Discriminator		An enumerated attribute that indicates the subclass that applies for each superclass occurrence.
Abstract class				A class with no direct occurrences.
Concrete class				A class that can have direct occurrences.

Alternate keys

- An **alternate key** is a candidate key that is not chosen as a primary key. Therefore each candidate key is either a primary key or an alternate key
- The UML has no specified notation for unique keys (i.e. alternate keys)
- It is possible to use the same notation used by IE, the **AKn.m** notation (see figure above)
  - AKn.m = column m^{th} of the n^{th} Alternate Key

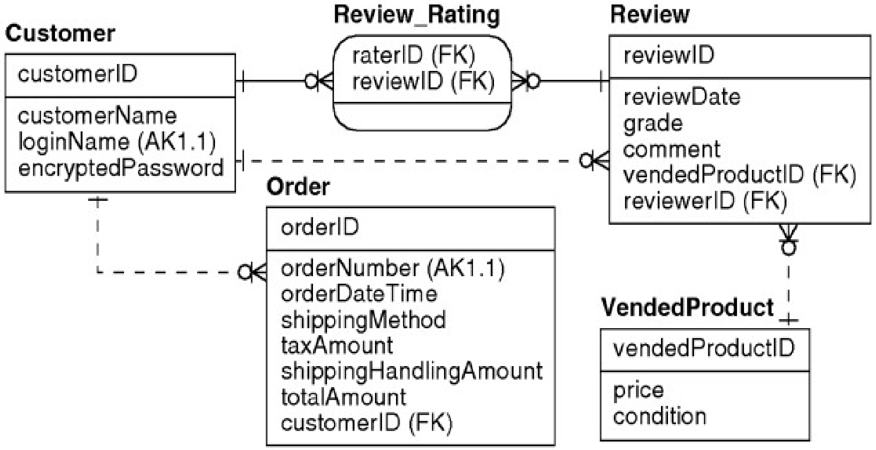


Surrogate key vs Natural key (1)

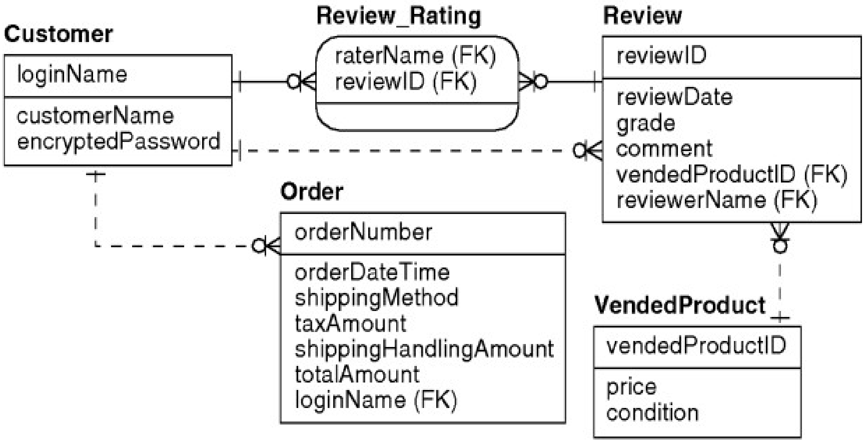
- **With existence-based identity each class has a generated identifier (also called a surrogate key) as its primary key. Each association has a primary key composed of identifiers from the related classes**
  - The advantage of this approach is that each class's primary key is a single attribute (often defined as a number)
  - Furthermore, since the primary key is synthetic, it is immutable
- **Another approach is value-based identity — a unique combination of real-world attributes (also called a natural key) identifies each class occurrence. "Real-world attributes" are those that come from the business problem description**
  - A downside is that the value of real-world attributes can change — such changes must propagate to foreign keys
  - Some models have a series of dependent entity types that lead to unwieldy multi-attribute primary keys
- **Unless there are unusual circumstances, it is recommend the use of surrogate keys (existence-based identity).**

Surrogate key vs Natural key (2)

Surrogate Key Example

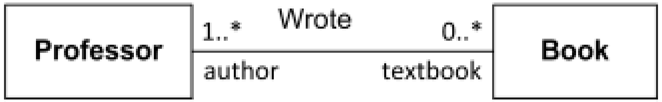


Natural Key Example

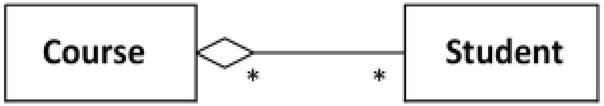


Association, Aggregation, Composition

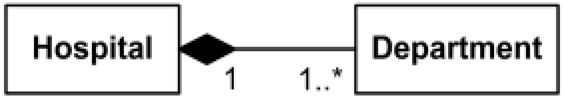
- **Association is a structural relationship that represents objects can be connected or associated with another object inside the system**



- **Aggregation and Composition are subsets of Association. In both object of one class "owns" object of another class:**
  - **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Course (parent) and Student (child). Delete the Course and the Students still exist.

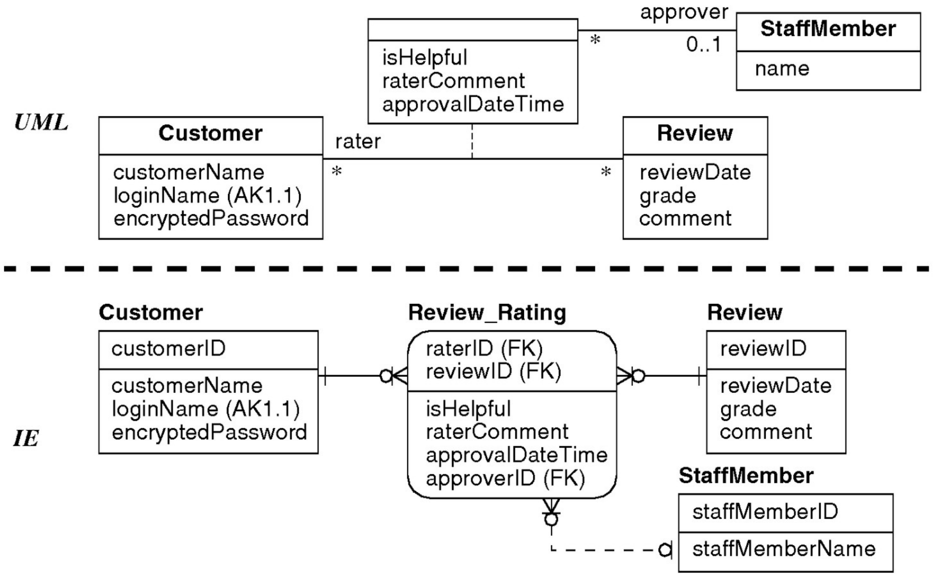


  - **Composition** implies a relationship where the child cannot exist independent of the parent. Example: Hospital (parent) and Department (child). Departments don't exist separate to a Hospital.



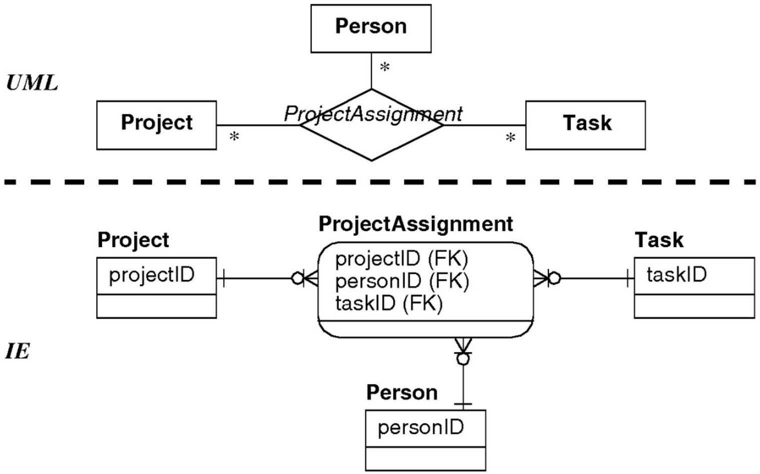
Association class

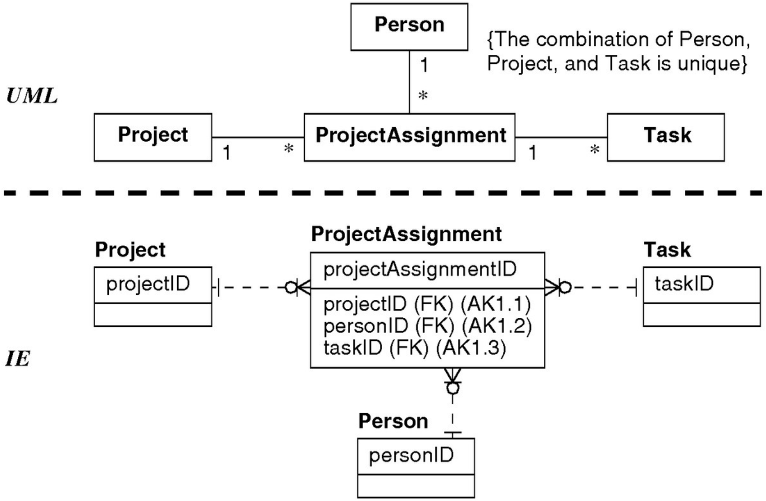
- An association class is an association that is also a class. Like the links of an association, the occurrences of an association class derive identity from the related objects
- Like a class, an association class can have attributes, operations, and associations
- The UML notation for an association class is a box that connects to the line corresponding association with a dotted line



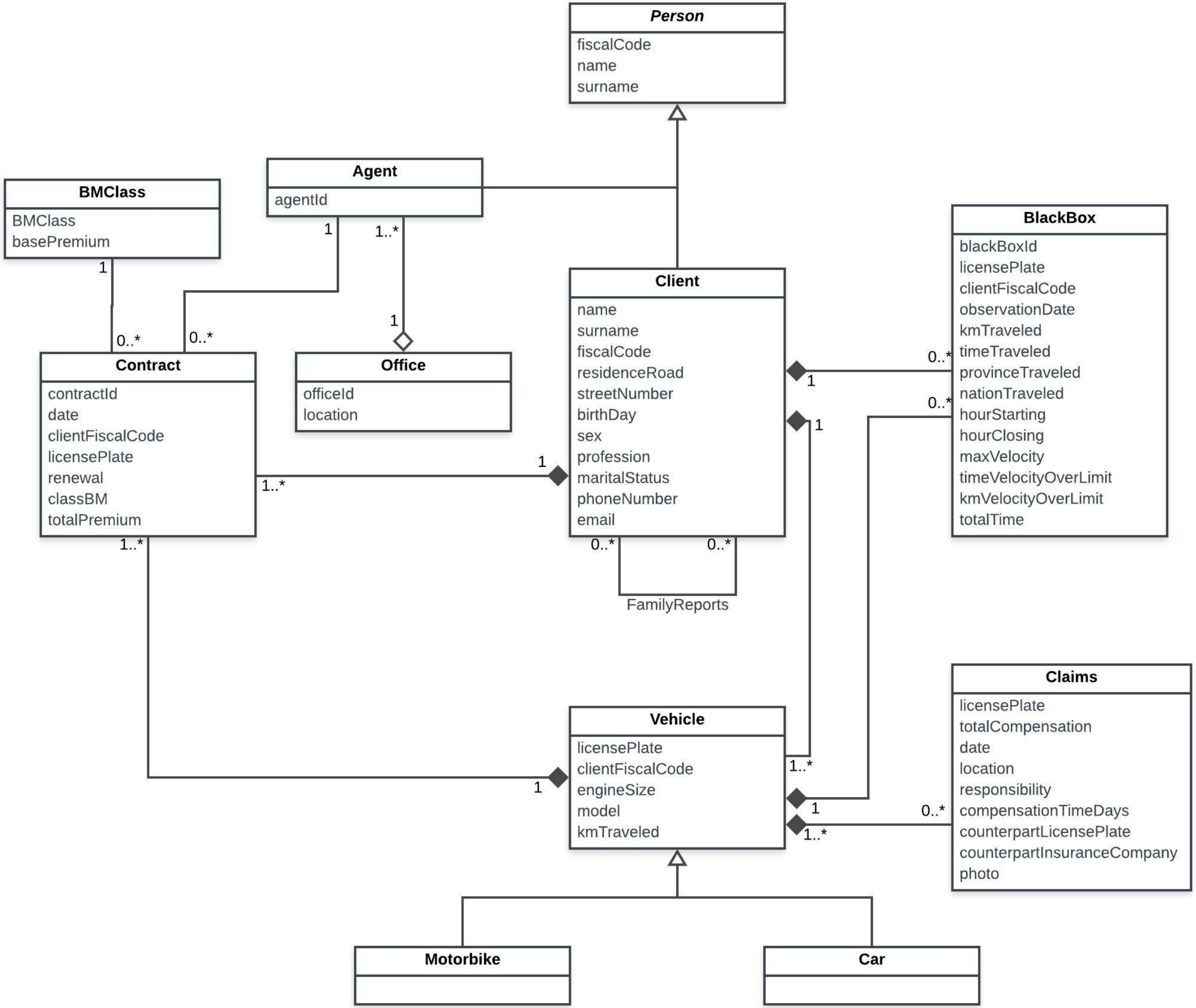
Ternary associations

- A ternary association is an association involving three classes
- The UML notation is a diamond with lines connecting the related classes
- Many supposed "ternary" associations are not fundamental and can be decomposed into binary associations, with possible qualifiers and attributes





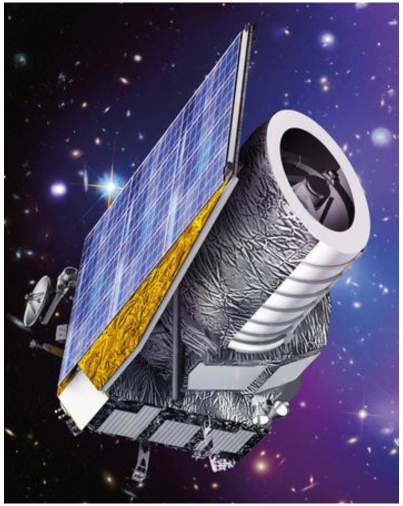
UML Example Insurance Company



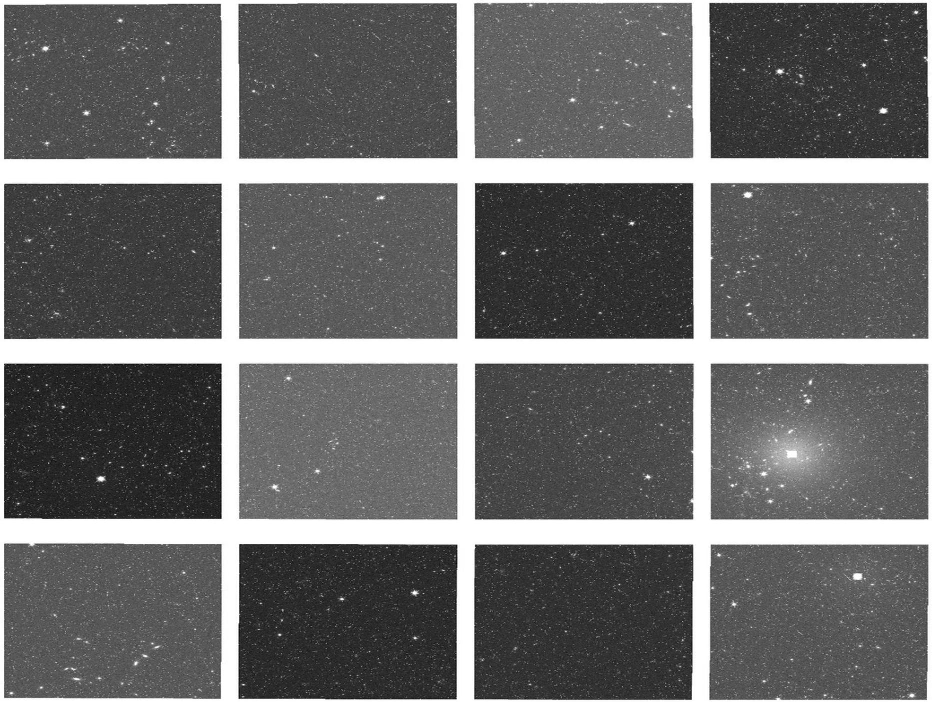
Credit to Andrea Pesce

Data model case study: Euclid

- **M2 mission in the framework of ESA Cosmic Vision Program**
- Euclid mission objective is to map the geometry and understand the nature of the **dark Universe** (dark energy and dark matter)
- **Federation** of 8 European + 1 US Science Data Centers and a Science Operation Center (ESA)
- Large amount of data produced by the mission
  - Due to reprocessing
  - Large amount of **external data** needed (ground based observations)
  - Grand total: **90 PB**
- Two instruments on board:
  - VIS: Visible Imager
  - NISP: Near Infrared Spectro-Photometer



A NISP instrument simulated image

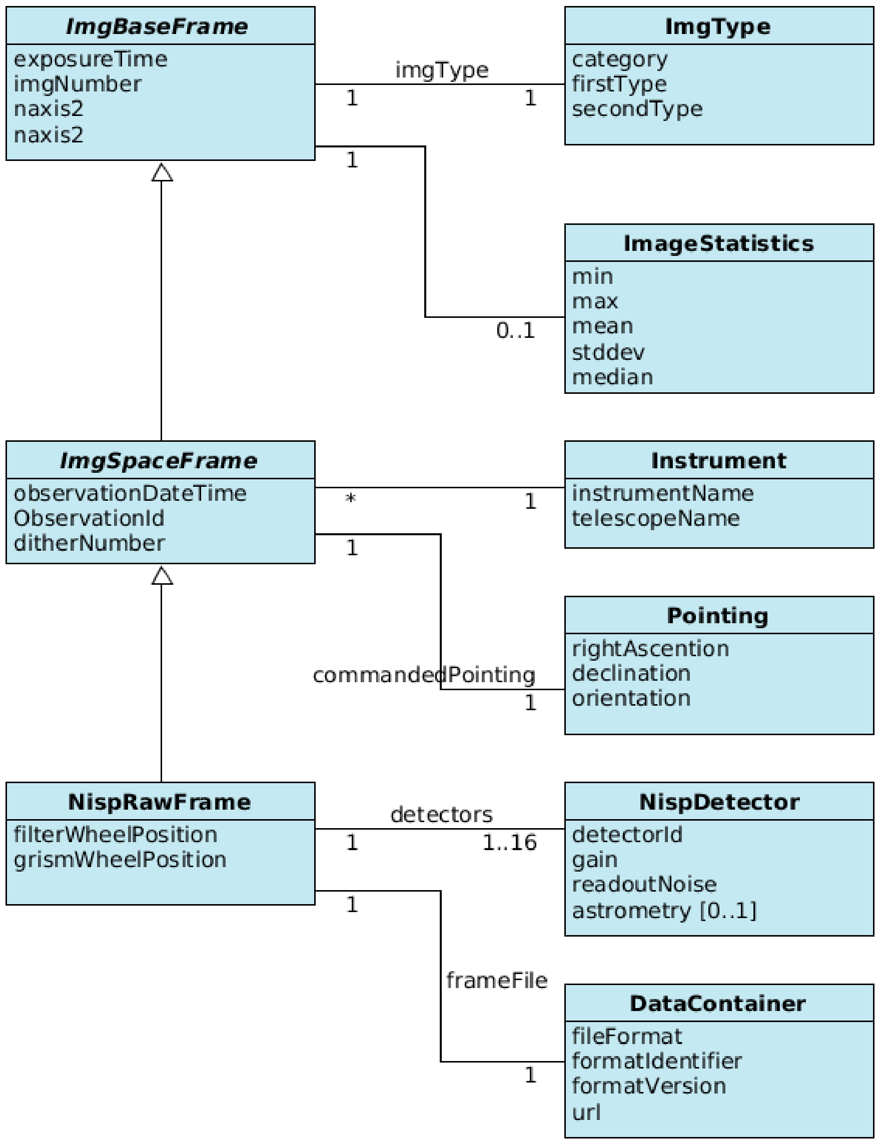


- The NISP focal plane is composed of a matrix of 4×4 2040×2040 18 micron pixel detectors
- The photometric channel is equipped with 3 broad band filters (Y, J and H)
- The spectroscopic channel is equipped with 4 different low resolution near infrared grisms (three red and one blue) but no slit
- The three red grims provides spectra with three different orientations (0°, 90°, 180°)

Metadata content (simplified)

- **We need to define the metadata associated to a NISP image (a single exposure)**
- **All images have a common set of information**
  - **Exposure time**, **image category** and purpose (is it a simulation, a calibration image, a sky image, etc.) and image **dimensions**, some **statistics** on the image, to quickly check if there are anomalies, and we need to keep the information about the **instrument** used to acquire a given image
  - Space telescopes can perform **surveys** of the sky, hence the observation can be identified by the **observation ID**. Moreover, for a given field, they can execute a **dithering** pattern, in order to increase the signal-to-noise ratio and reduce cosmic-ray hits. So we need also to store the dither number. Additional information needed are the **observation date and time** and the **commanded pointing** (right ascension, declination and telescope orientation)
- Then we have information specific to the Euclid instruments. The NISP instrument has both a filter wheel and a grism wheel. The images from **all detectors** should be stored in a single file, to simplify its retrieval and the analysis. However, each detector has some specific properties: **gain**, **readout noise**. Then, for each detector we need to compute the mapping from pixel indexes to sky coordinates (RA, DEC), i.e. its own **astrometric solution**.

UML Example Euclid



Credit to Marco Frailis

Are you still there?
Questions?

Data model implementation

- **Once we have designed a data model in UML, we need to convert the diagrams into machine readable formats**
  - To perform additional validations to the data model, e.g. homogeneity, common naming rules
  - To be able to persist objects and relations which are compliant with the designed data model
- **The implementation depends on the underlying technology:**
  - For relational databases: database schema
  - For document oriented databases: the XML Schema Language (XSD) or the JSON Schema (JavaScript Object Notation)
- **Document based systems can also be built on top of relational databases**
- **Remember also that UML can be used also for software design modeling**

Database systems

- **Several criteria can be used to classify Database Management Systems (DMBS)**
- **One of these criteria is the data model used:**
  - **Relational DBMSs**: they use the relational model, which represents a database as a collection of tables, where each table can be stored as a separate file. Most relational databases use the high-level query language called **SQL**
  - **NoSQL databases**:
    - Document based NoSQL sytems: data in form of documents using well-known formats, such as JSON, accessed by ID or indexes
    - NoSQL key-value stores: simple data model based on fast access by the key to the value associated with the key
    - Column-based NoSQL systems: partition a table by column into column families, where each column family is stored in its own files
    - Graph-based NoSQL systems: Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions
  - Hybrid systems: e.g. **XML databases**

Relational database system

- **Relational model: represents the database as collection of *relations***
  - Each relation represents a **table** of values, i.e. a flat file of records
  - Each row in the table represents a collection of related data values
  - A row represents a fact that typically corresponds to a real-world **entity** or **relationship**
  - Table name and column names are used to interpret the meaning of the values in each row
- **A relation schema R, denoted by $R(A_1, A_2, \ldots, A_n)$ is made up of a relation name R and a list of attributes $A_1, A_2, \ldots, A_n$.**
  - The attribute A_i is the name of a role played by some domain D in the relation schema R
  - The degree of the relation R is the number of attributes *n*
  - D is called domain of A_i and denoted by **dom(A_i)**
  - A **domain** is given: name, **data type** and **format**

Relation (instance)

- **A relation** (or **relation state**) **r of a relation schema is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$ and is denoted as *r(R)***
- **Each n-tuple t is an ordered list of n values $t = \langle v_1, v_2, \ldots, v_n \rangle$, where each value v_i is an element of $dom(A_i)$ or a special NULL value**
  - Each value in a tuple is an **atomic** value (not divisible into components)
  - **We can have several meanings for NULL values**: value unknown, value exists but not available, attribute does not apply (i.e. value indefined)
- A relation *r(R)* is a mathematical relation of degree n on the domains $dom(A_1)$, $dom(A_2)$, …, $dom(A_n)$, which is a subset of the **Cartesian product** of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times \cdots \times dom(A_n))$$

- Some relations may represent facts about **entities**, other relations may represent facts about **relationships**

Relational model constraints

- **Inherent model-based constraints** (or implicit constraints)
  - E.g.: a relation cannot have duplicate tuples
- **Schema-based constraints** (or explicit constraints)
  - Typically directly expressed in the schema of the data model, using a Data Definition Language (DDL)
  - E.g.: domain constraints, key constraints (see next slides)
- **Application-based or semantic constraints**
  - Constraints that cannot be directly expressed in the schema of the data model
  - They must be enforced by application programs
- For each attribute, a constraint can specify whether NULL values are or are not permitted

Domain and key constraints

- **Domain constraints specify that each value of each attribute A must be an atomic value from the domain dom(A)**
  - Data types associated with domains typically include standard numeric types for integers, real numbers, characters, booleans, fixed-length and variable length strings, date, time, etc.
- **A subset of attributes of the relational schema R is called superkey (SK) of R if for any two distinct tuples t_1 and t_2 of a relation state *r* of *R*, $t_1[SK] \neq t_2[SK]$**
  - Every relation has at least one default SK, the set of all its attributes
- **A candidate key (CK) of a relational schema R is a SK of R with the additional property that removing any attribute from CK leaves a set of attributes that is not a superkey of R**
  - A relation schema may have more than one CK
- A **primary key** (PK) is a CK whose values are used to *identify* tuples in the relation
  - It is usually better to choose a PK with a single attribute or a small number of attributes
  - A PK composed of one column is called single primary key, a combination of column is called composite primary key
  - The other candidate keys are designated as **unique keys** (or **alternate keys**)

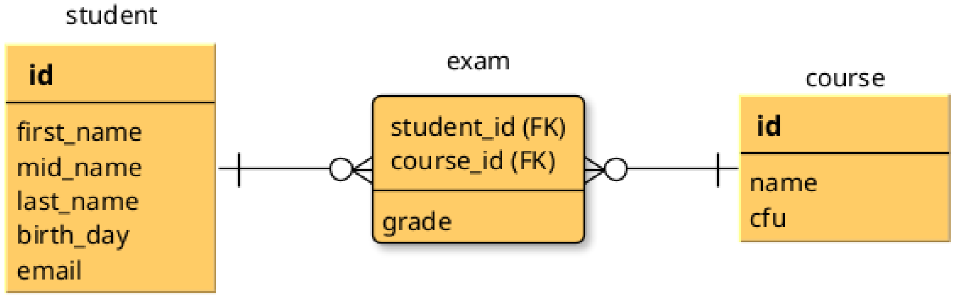
Entity and Referential Integrity Constraints

- **The entity integrity constraints states that no primary key value can be NULL**
  - The primary key is used to identify individual tuples in the relation
- **A referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples of the two relations**
  - A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:
    1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2.
    2. A value of FK in a tuple t_1 in the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is NULL. In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 references the tuple t_2
  - If the two conditions above hold between R_1 and R_2, a **referential integrity constraint** from R_1 to R_2 is said to hold

Relational database

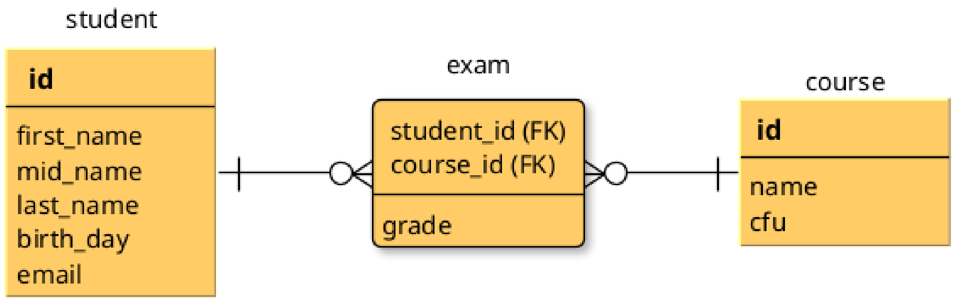
- A relational database usually contains many relations
- A **relational database schema** S is a set of relation schemas S = {R_1, R_2, ..., R_m} and a set of **integrity constraints** (IC)
- A relational database **state** DB of S is a set of relation states DB = {r_1, r_2, ..., r_m} such that r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC
- A database state that does not obey all the integrity constraints is called an **invalid state**, and a state that satisfy all the constraints in the defined set of integrity constraints IC is called a **valid state**
- Each relational DBMS must have a **data definition language** (DDL) for defining a relational database schema
  - Current relational DBMS-s are mostly using the SQL language for this purpose

Relational model example (1)



Relational model example (2)

```sql
CREATE TABLE student(
id          INTEGER PRIMARY KEY AUTOINCREMENT,
first_name  VARCHAR(20) NOT NULL,
mid_name    VARCHAR(20),
last_name   VARCHAR(20) NOT NULL,
birth_day   DATE NOT NULL,
email       VARCHAR(20) UNIQUE NOT NULL);

CREATE TABLE course(
id   INTEGER PRIMARY KEY AUTOINCREMENT,
name VARCHAR(20) UNIQUE NOT NULL,
cfu  INTEGER NOT NULL);

CREATE TABLE exam(
student_id INTEGER,
course_id  INTEGER,
grade      INTEGER NOT NULL,
FOREIGN KEY(student_id) REFERENCES student(id),
FOREIGN KEY(course_id) REFERENCES course(id));
```




```
-- SQLite tips

PRAGMA foreign_keys = ON;
.mode column
.header on
```

Relational model example (3)

```
sqlite> SELECT * FROM student;
id          first_name  mid_name    last_name   birth_day   email
----------  ----------  ----------  ----------  ----------  ---------------
0           Luca                    Rossi       1990-12-01  rossi@email.com
1           Luca                    Rossi       1985-05-05  rossi@email.it
2           Maria       Grazia      Bianchi     1985-05-05  maria@email.com
```

```
sqlite> SELECT * FROM course;
id          name                  cfu
----------  --------------------  -----
0           Open Data Management  6
1           Machine Learning      8
```

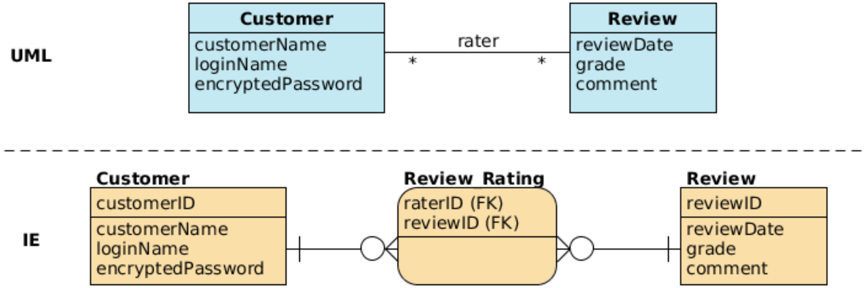
```
sqlite> SELECT * FROM exam;
student_id  course_id   grade
----------  ----------  ----------
0           1           25
1           0           28
1           1           30
```

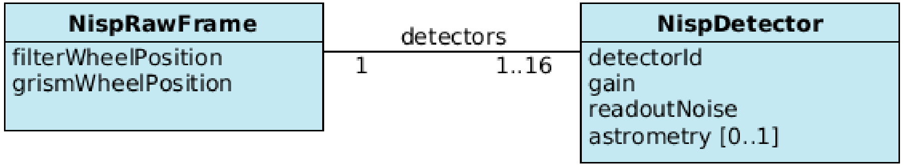
Object-relational impedance mismatch

- **A set of conceptual and technical difficulties that are often encountered when a relational database management system is been served by an application program written in an object oriented language**
- **We have already discussed some solutions when comparing the UML model with the IE model in the previous lecture**
- **Additional difficulties:**
  - Hierarchical structure:
    - In UML, we can define complex hierarchical structures. A class can "aggregate" instances of other classes. The relational model only "accepts" atomic types for the entity attributes and relations
    - In the relational model, children point to their parent, while in the hierarchical model parents point to their children
  - Inheritance:
    - Not directly supported by the relational model. Several mappings can be implemented to keep the inheritance information
  - Class normalization vs data normalization

Examples

- Many-to-many associations, when mapped to a relational schema, require an additional table, i.e. an additional relation
- In the relational schema we cannot define an upper limit on the multiplicity
- Abstract classes have multiple mapping options, each one with some limitations



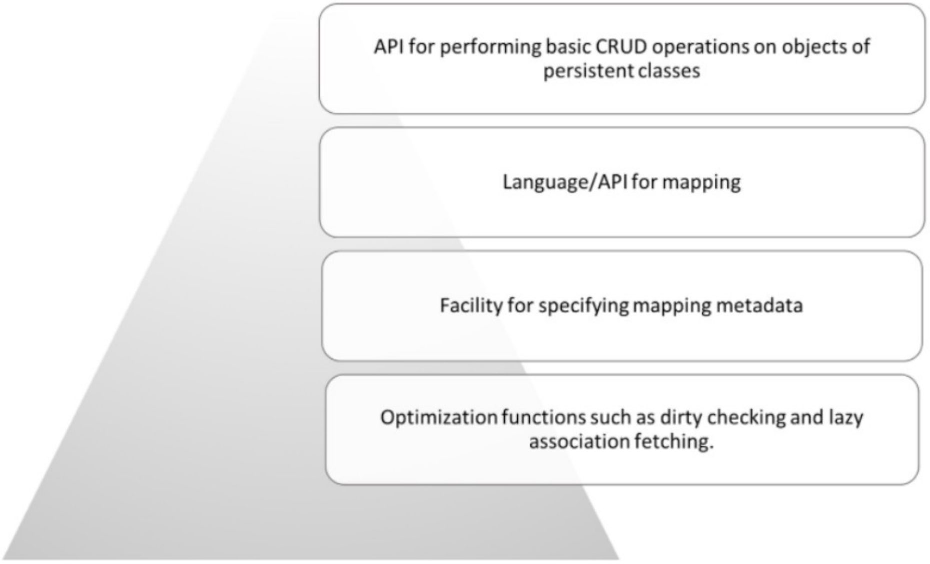


Specialization and generalization

- We consider here only the single inheritance
- To convert each specialization with m subclasses $\{S_1, S_2, \ldots, S_m\}$ and superclass C, where the attributes of C are $\{k, a_1, a_2, \ldots, a_n\}$ and k is the primary key, into a relation schema, the options are:
  - **Multiple relations - superclass and subclasses**. Create a relation L for C with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, with attributes Attrs(L_i) = $\{k\} \cup \{$attributes of $S_i\}$ and PK(L_i) = k.
  - **Multiple relations – subclass only**. Create a relation L_i for each subclass S_i, with the attributes Attrs(L_i) = $\{$attributes of $S_i\} \cup \{k, a_1, \ldots, a_n\}$ and PK(L_i) = k.
  - **Single relation with one type attribute**. Create a single relation schema L with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\} \cup \{$attributes of $S_1\} \cup \ldots \cup \{$attributes of $S_m\} \cup \{t\}$ and PK(L) = k. The attribute t is called type (or discriminating) attribute whose value indicates the subclass to which each tuple belongs
  - **Single relation multiple type attributes**. As above, but instead of a single type attribute t, there is a set $\{t_1, t_2, \ldots, t_m\}$ of m boolean type attributes indicating wether or not a tuple belongs to subclass S_i.

Object-Relational Mapping (ORM)

- **Object-relational mapping (ORM)** uses **different tools, technologies and techniques to map data objects in a target programming language to relations and tables of a RDBMS**
- **An ORM solution consists of the following four pieces:**



ORM solutions

- **An ORM abstracts your application away from the underlying SQL database and SQL dialect**
- **If the tool supports a number of different databases (and most do), this confers a certain level of portability on your application**
- **Several programming languages have at least one ORM solution**
  - Java: it provides both a standard specification, named Java Persistence API (JPA), and several implementations of the spefication (Hibernate, EclipseLink)
  - C++: possible ORM solutions are
    - ODB: https://www.codesynthesis.com/products/odb
    - QxOrm: https://www.qxorm.com/qxorm_en/home.html
  - Python:
    - **SQLAlchemy**: https://www.sqlalchemy.org/
    - The **Django** framework: https://docs.djangoproject.com/en/2.1/topics/db/
    - Pony: https://ponyorm.com/
  - Ruby: ActiveRecord, DataMapper, Sequel

SQLAlchemy (1)

- **The SQLAlchemy SQL Toolkit and Object Relational Mapper is a comprehensive set of tools for working with databases and Python**
- **It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access**
- **SQLAlchemy has dialects for many popular database systems including Firebird, Informix, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, or Sybase**
- **The SQLAlchemy has four ways of working with database data:**
  - Raw SQL
  - SQL Expression Language
  - Schema Definition Language
  - ORM

SQLAlchemy (2)

- SQLAlchemy ORM consists of several components
  - **Engine**
    - It manages the connection with the database
    - It is created using the create_engine() function
  - Declarative **Base** class
    - It maintains a catalog of classes and tables
    - It is created using DeclarativeBase and is bound to the engine
  - **Session** class
    - It is a container for all conversations with the database
    - It is created using the sessionmaker() function and is bound to the engine
- https://docs.sqlalchemy.org/en/20/tutorial/index.html

Prerequisites

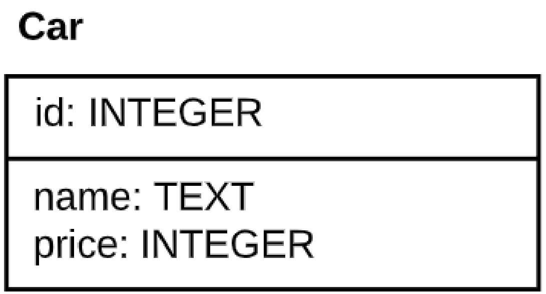
- **Download and install the Python Anaconda (or Miniconda) Distribution, with Python version 3.x: https://www.anaconda.com/download**
- **Then you need to install some additional python packages for the following exercise/hands-on:**
  - To install the Django framework use the following command line:

```
conda create -n orm_sqlalchemy sqlalchemy
conda activate orm_sqlalchemy
```

- Clone the GIT repository and enter the directory of SQLAlchemy examples

```
git clone https://www.ict.inaf.it/gitlab/bignamini/orm_project.git
cd orm_example/sqlalchemy_example
```

ORM with SQLAlchemy: Example 1



- Engines https://docs.sqlalchemy.org/en/20/core/engines.html
- Declarative Base https://docs.sqlalchemy.org/en/20/orm/declarative_styles.html
- Session https://docs.sqlalchemy.org/en/20/orm/session.html
- Query https://docs.sqlalchemy.org/en/20/orm/queryguide/query.html

ORM with SQLAlchemy: Example 1

Car

id: INTEGER
name: TEXT price: INTEGER

create.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Base
from mapping import engine

Base.metadata.create_all(bind=engine)

```

mapping.py

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import sessionmaker

from sqlalchemy import Column, Integer, String


# Create a new Engine instance.
engine = create_engine('sqlite:///example_1.db')


# Construct a base class for declarative class definitions
class Base(DeclarativeBase):
    pass

# Declarative mapping for Car
class Car(Base):
    __tablename__ = "car"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    price = Column(Integer)


# Create a configurable Session factory.
Session = sessionmaker(bind=engine)

```

ORM with SQLAlchemy: Example 1

insert.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Car
from mapping import Session

# Create a Session object
session = Session()

# Add instances of Car objects to the session
session.add_all(
   [Car(id=1, name='Audi', price=52642),
    Car(id=2, name='Mercedes', price=57127),
    Car(id=3, name='Skoda', price=9000),
    Car(id=4, name='Volvo', price=29000),
    Car(id=5, name='Bentley', price=350000),
    Car(id=6, name='Citroen', price=21000),
    Car(id=7, name='Hummer', price=41400),
    Car(id=8, name='Volkswagen', price=21600)])

# Commit changes to database
session.commit()

# Close session
session.close()
```

read.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Car
from mapping import Session

# Create a Session object
session = Session()

# Query all the cars
results = session.query(Car).all()

# Print the results
for car in results:
    print("The price of", car.name, "is", car.price)

# Close session
session.close()
```

```
$ python read.py

The price of Audi is 52642
The price of Mercedes is 57127
The price of Skoda is 9000
The price of Volvo is 29000
The price of Bentley is 350000
The price of Citroen is 21000
The price of Hummer is 41400
The price of Volkswagen is 21600
```

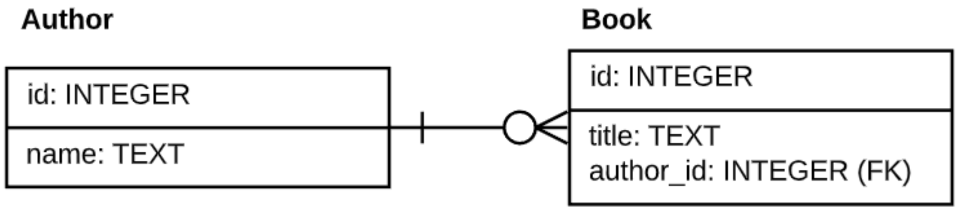
ORM with SQLAlchemy: Example 1

filter.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Car
from mapping import Session
from sqlalchemy.sql import and_

# Create a Session object
session = Session()

# Query cars with name ending with 'en'
results = session.query(Car).filter(Car.name.like('%en'))

# Print the results
print("Cars with name ending with 'en' are:")
for car in results:
    print(car.name)
print()

# Query cars filtered by id
results = session.query(Car).filter(Car.id.in_([2, 4, 6, 8]))

# Print the results
print("Cars with id in [2, 4, 6, 8] are:")
for car in results:
    print(car.id, car.name)
print()

# Query cars filtered by price
results = session.query(Car).filter(and_(Car.price > 10000,
                                         Car.price < 40000))

# Print the results
print("Cars with price between 10000 and 40000 are:")
for car in results:
    print(car.name, car.price)
```

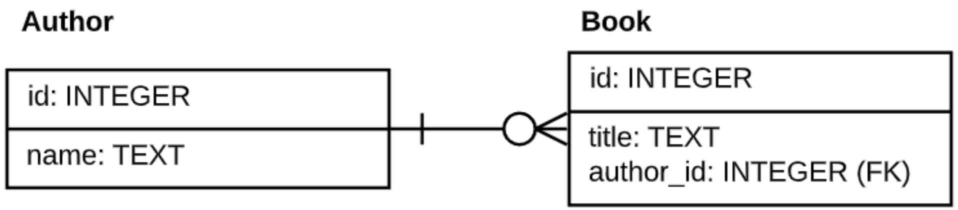
```
$ python filter.py

Cars with name ending with 'en' are:
Citroen
Volkswagen

Cars with id in [2, 4, 6, 8] are:
2 Mercedes
4 Volvo
6 Citroen
8 Volkswagen

Cars with price between 10000 and 40000 are:
Volvo 29000
Citroen 21000
Volkswagen 21600
```

ORM with SQLAlchemy: Example 2



- Foreign keys in SQLite
  https://docs.sqlalchemy.org/en/20/dialects/sqlite.html#foreign-key-support
- Relationship
  https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html

ORM with SQLAlchemy: Example 2



mapping.py

```python
# Declarative mapping for Author and Book classes
class Author(Base):
    __tablename__ = "author"

    id = Column(Integer, primary_key=True)
    name = Column(String)

    book = relationship("Book")

class Book(Base):
    __tablename__ = "book"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    author_id = Column(Integer, ForeignKey("author.id"))

    author = relationship("Author")

# Enable foreign key constraint in SQLite
@event.listens_for(Engine, "connect")
def _set_sqlite_pragma(dbapi_connection, connection_record):
    if isinstance(dbapi_connection, SQLite3Connection):
        cursor = dbapi_connection.cursor()
        cursor.execute("PRAGMA foreign_keys=ON;")
        cursor.close()
```

ORM with SQLAlchemy: Example 2

insert.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Author, Book
from mapping import Session

# Create a Session object
session = Session()

# Add instances of Author and Book objects to the session
session.add_all(
   [Author(id=1, name='Lev Tolstoy'),
    Author(id=2, name='Jane Austen'),
    Author(id=3, name='Charles Dickens'),
    Book(id=1, title='War and Peace', author_id=1),
    Book(id=2, title='Anna Karenina', author_id=1),
    Book(id=3, title='Emma', author_id=2),
    Book(id=4, title='David Copperfield', author_id=3)])

# Commit changes to database
session.commit()

# Close session
session.close()
```

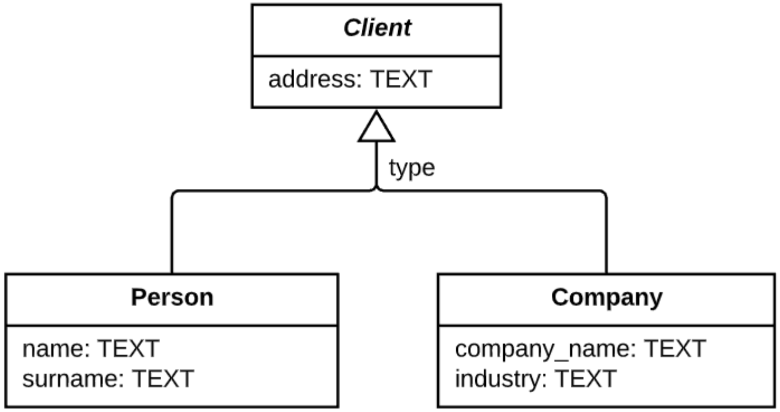
filter.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Author, Book
from mapping import Session

# Create a Session object
session = Session()

# Query Author to select Lev Tolstoy
results = session.query(Author).filter(Author.name=="Lev Tolstoy").first()

# Print all books of Lev Tolstoy
print('Books of Lev Tolstoy are:')
for book in results.book:
    print(book.title)
print()

# Query Book for the book Emma and get its author
results = session.query(Book).filter(Book.title=="Emma").first()

print('The author of', results.title, 'is', results.author.name)

# Close session
session.close()
```

```
$ python filter.py

Books of Lev Tolstoy are:
War and Peace
Anna Karenina

The author of Emma is Jane Austen
```

Inheritance in Python

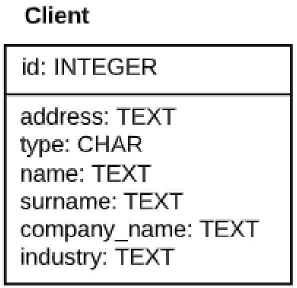
- This is a simple example of inheritance in UML and how can be implemented in Python



UML

```python
class Client(object):
    """docstring for Client"""

    def __init__(self, address):
        super(Client, self).__init__()
        self.address = address

class Person(Client):
    """docstring for Person"""

    def __init__(self, name, surname, address):
        super(Person, self).__init__(address)
        self.name = name
        self.surname = surname

class Company(Client):
    """docstring for Company"""

    def __init__(self, company_name, industry, address):
        super(Company, self).__init__(address)
        self.company_name = company_name
        self.industry = industry
```

Inheritance in a Relational Database



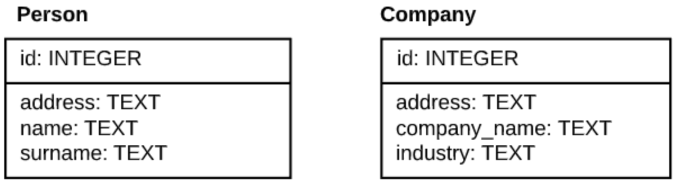
IE

Single table inheritance

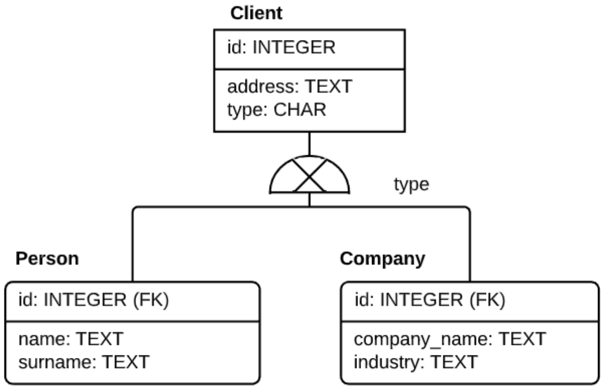
- Unique ID
- No JOIN necessary
- Many NULL attributes

Concrete table inheritance

- Not unique ID
- No JOIN necessary
- No NULL attributes



IE

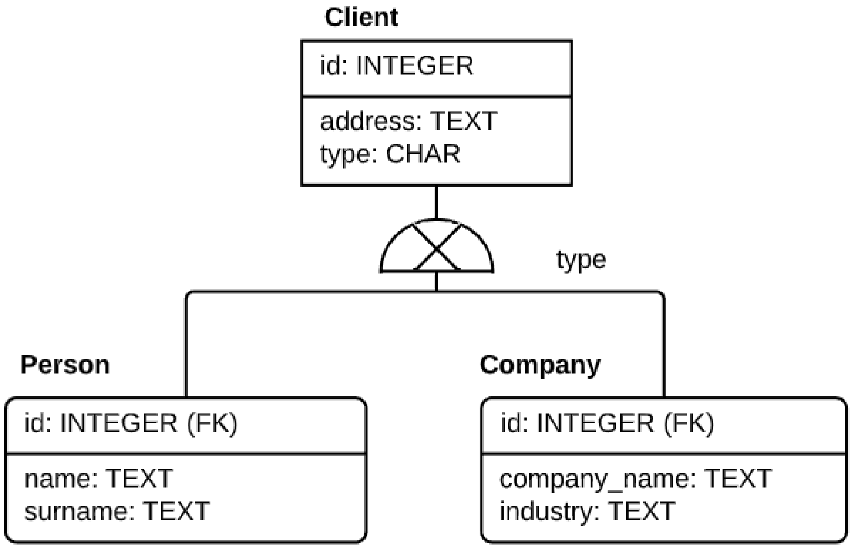


IE

Joined table inheritance

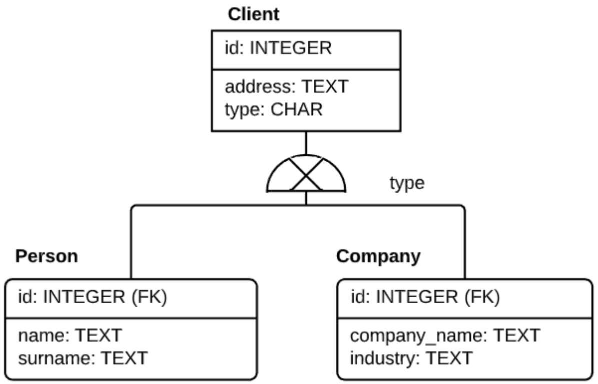
- Unique ID
- JOIN necessary
- No NULL attributes

ORM with SQLAlchemy: Example 3



- Inheritance
  https://docs.sqlalchemy.org/en/20/orm/inheritance.html

ORM with SQLAlchemy: Example 3



mapping.py

```python
class Base(DeclarativeBase):
    pass

# Declarative mapping for Client
class Client(Base):
    __tablename__ = 'client'

    id = Column(Integer, primary_key=True)
    address = Column(String)
    type = Column(String)

    __mapper_args__ = {
        'polymorphic_identity':'client',
        'polymorphic_on':type
        }

# Declarative mapping for Person
class Person(Client):
    __tablename__ = 'person'

    id = Column(Integer, ForeignKey('client.id'), primary_key=True)
    name = Column(String)
    surname = Column(String)

    __mapper_args__ = {
        'polymorphic_identity':'person',
    }

# Declarative mapping for Company
class Company(Client):
    __tablename__ = 'company'

    id = Column(Integer, ForeignKey('client.id'), primary_key=True)
    company_name = Column(String)
    industry = Column(String)

    __mapper_args__ = {
        'polymorphic_identity':'company',
    }
```

ORM with SQLAlchemy: Example 3

insert.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Client, Person, Company
from mapping import Session

# Create a Session object
session = Session()

# Add instances of Person and Company objects to the session
session.add_all([
    Person(name='Mario', surname='Rossi', address='via Giulia'),
    Person(name='Luigi', surname='Bianchi', address='via Flavia'),
    Company(company_name='Acegas', industry='multi-utility', address='via del Teatro'),
    Company(company_name='Illy', industry='coffee', address='via Flavia')
    ])

# Commit changes to database
session.commit()

# Close session
session.close()
```

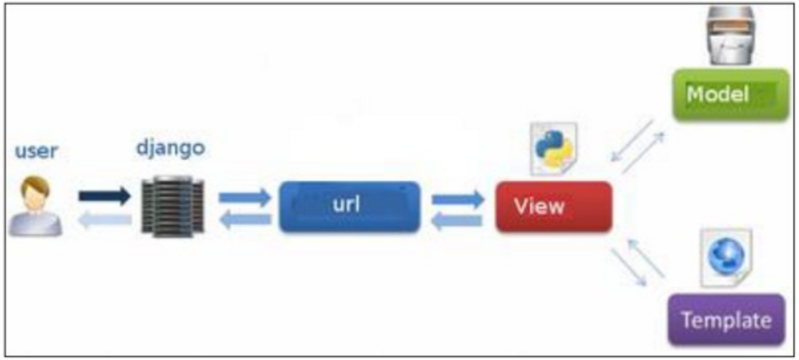
filter.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mapping import Client, Person, Company
from mapping import Session

# Create a Session object
session = Session()

# Query Client with address via Flavia
results = session.query(Client).filter(Client.address=='via Flavia')

# Print results
print('Clients in via Flavia are:')
for client in results:
    if client.type == 'person':
        print(client.id, client.name, client.surname)
    elif client.type == 'company':
        print(client.id, client.company_name, client.industry)
print()

# Query Person with address via Flavia
results = session.query(Person).filter(Person.address=='via Flavia')

# Print results
print('Persons in via Flavia are:')
for client in results:
    print(client.id, client.name, client.surname)

# Close session
session.close()
```

```
$ python filter.py

Clients in via Flavia are:
2 Luigi Bianchi
4 Illy coffee

Persons in via Flavia are:
2 Luigi Bianchi
```

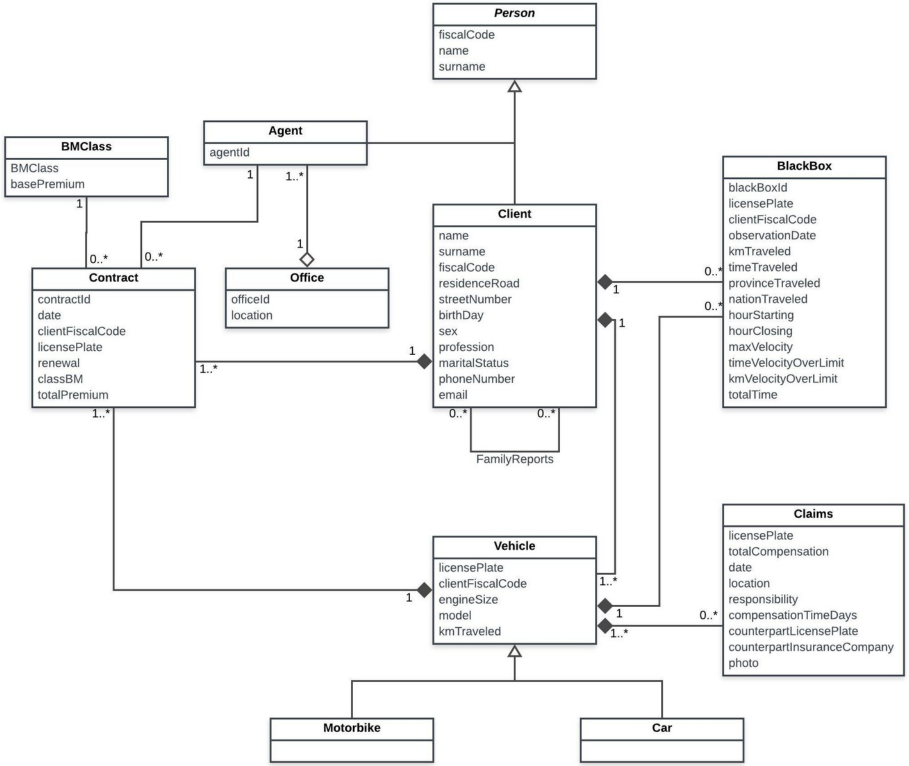
Django

- Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design
  https://www.djangoproject.com
- Django follows the model-template-view (MTV) architectural pattern
  - An object-relational mapper, defining a data model as python classes (**Models**)
  - A system for processing HTTP requests (**Views**) with a web templating sytem (**Template**)
  - A regular-expression-based URL dispatcher (**Url**)



- Django comes with a lightweight standalone web server for development and testing
- A serialization system that can produce and read XML and/or JSON representation of Django models
- Lot of reusable packages provided by the community:
  https://djangopackages.org/

Data Model for Insurance Company



Credit to Andrea Pesce

Prerequisites

- The simplest way to install Django is to download and install the Python Anaconda Distribution, with Python version 3.x: https://www.anaconda.com/download
- Then you need to install some additional python packages for the following exercise/hands-on:
  – To install the Django framework use the following command line:

```
conda create -n insurance django
```

  – Additional packages are needed, not available in Anaconda but installed with the “pip” command:

```
pip install django-extensions djangorestframework
pip install django-composite-field django-url-filter
pip install django-phonenumber-field phonenumbers
pip install Pillow
```

ORM project example

- The entire example can be retrieved at the following link:

  https://www.ict.inaf.it/gitlab/odmc/orm_example

- You can clone the project with the git version control system, i.e. with the command:

```
git clone https://www.ict.inaf.it/gitlab/bignamini/orm_project.git
cd orm_example/django_example
```

- Anyway, to create a Diango project from scratch you can use the following commands

```
django-admin startproject insurance
cd insurance
python manage.py startapp insurancedb
```

  which creates a project folder, named **insurance**, with additional files and then an application, named **insurancedb**, inside the project.
  It automatically creates skeleton files needed by a Django project and application

Project structure

```
insurance/
├── insurancedb
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── insurance
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

File configuring the Admin site

File containing the app data model

Views on the data model classes

Project settings: app list and configuration

Site urls declaration

- For admin.py, models.py, urls.py and views.py files we are going to use the ones in the git repository
- We must edit the settings.py

Django Implementation (1)

- Each class inherits models.Model
- All fields use a Django Model Data Type https://www.webforefront.com/django/modeldatatypesandvalidation.html
  - models.CharField(max_length = 20)
  - models.BooleanField()
  - models.FloatField()
  - models.DateTimeField()
  - …
- Attributes in the Data Model Type are used to set options for fields
  - null = True
  - primary_key = True
- Foreign keys https://docs.djangoproject.com/en/1.11/ref/models/fields/#django.db.models.ForeignKey

```
licensePlate = models.ForeignKey(Vehicle)
```

- Related names https://docs.djangoproject.com/en/dev/topics/db/queries/#backwards-related-objects

```
fiscalCode1 = models.ForeignKey(Client, on_delete = models.CASCADE, related_name = "primo")
```

Django Implementation (2)

- **By enumerated type** we mean a type that provides a set of possible values through the **choices** parameter (option) available to all field types

```
FAMILY_REPORTS = ('primo', 'secondo', 'terzo')

relationship = models.CharField(max_length=7, choices = [(d,d) for d in FAMILY_REPORTS])
```

- Model Meta options is “anything that’s not a field”
  - Abstract class
  - Ordering
  - Candidate key of multiple columns
  - …

```
class Meta:
    Abstract = True

class Meta:
    Ordering = ['surname']

class Meta:
    unique_together = (("fiscalCode1", "fiscalCode2"),)
```

- It is a good practice to override the default name of objects

```
def __str__(self):
        return self.name
```

DB Schema creation

- Once we have defined our data model in **insurancedb/models.py** we need Django to create the corresponding DB schema
- First let's check the the project settings includes the imagedb application, i.e. that the file **insurance/settings.py** contains the the strings highlighted in red in the box on the bottom left
- To do the first migration, i.e. generation of the DB schema, **run the following command**

```
python manage.py makemigrations
```

output

```
Migrations for 'insurancedb':
  insurancedb/migrations/0001_initial.py
    - Create model BMClass
    - Create model Client
    - Create model Office
    - Create model Vehicle
    - Create model Contract
    - Create model Claims
    - Create model BlackBox
    - Create model Agent
    - Create model FamilyReports
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_extensions',
    'insurancedb',
    'rest_framework',
    'url_filter',
]
```

Then run the command

```
python manage.py migrate
```

Data insertion

- We can now open a python shell and interact with the data model API

```
python manage.py shell
```

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from insurancedb.models import BMClass

In [2]: bonus = BMClass(BMClass=1, basePremium=100.00)

In [3]: bonus.save()

In [4]: quit()
```

- You can pass a Python script to insert data

```
python manage.py shell < ../insert.py
```

Django urls.py and views.py

- A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations
- To design URLs for an app, you create a Python module informally called a URLconf (URL configuration). This module is pure Python code and is a mapping between URL path expressions to Python functions (your views)
- A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be:
  - HTML contents
  - A redirect
  - A 404 error
  - An XML document
  - An image
  - …

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Django admin.py

- Django provides an automatic admin interface
- It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site
- You can customize the admin interface editing the admin.py
- Setup an admin user

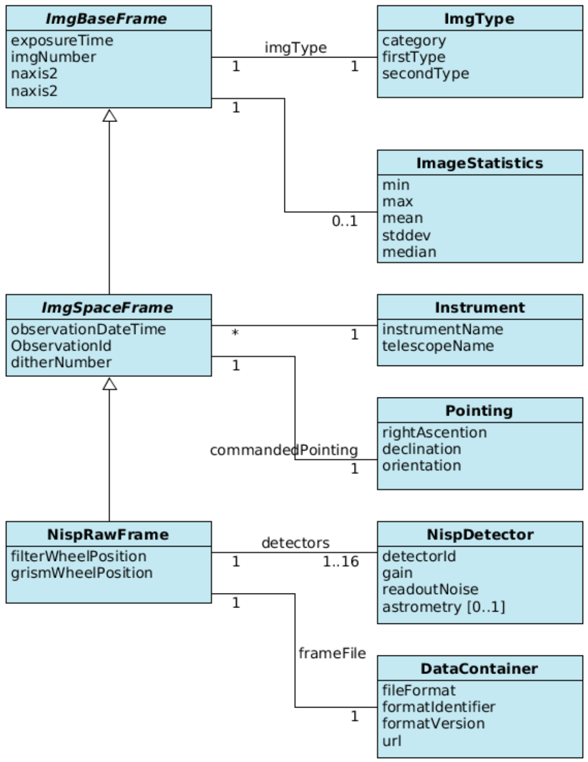
```
python manage.py createsuperuser
```

- Run the Django web server

```
python manage.py runserver
```

- Access to http://127.0.0.1:8000/

The NISP image data model



Credit to Marco Frailis

Implementation with Django

- **To implement the previous data model, in the following we will use the ORM provided by the Django web framework, in Python language**
- **Django features:**
  - An object-relational mapper, defining a data model as python classes (**Models**)
  - A system for processing HTTP requests with a web templating sytem (**Views**)
  - A regular-expression-based URL dispatcher (**Controller**)
  - A lightweight standalone web server for development and testing
  - A serialization sytsem thatn can produce and read XML and/or JSON representation of Django models
  - Lot of reusable packages provided by the community: https://djangopackages.org/
- **Several frameworks to build a REST API, e.g.: https://www.django-rest-framework.org/**

Prerequisites

- **The simplest way to install Django is to download and install the Python Anaconda Distribution, with Python version 3.x:** **https://www.anaconda.com/download**
- **Then you need to install some additional python packages for the following exercise/hands-on:**
  - To install the Django framework use the following command line:

```
conda install django
```

  - Additional packages are needed, not available in Anaconda but installed with the “pip” command:

```
pip install django-extensions djangorestframework django-composite-field
pip install django-url-filter
```

- **Another tool used, Jupyter, is already available in Anaconda**

ORM project example

- The entire example can be retrieved at the following link:
  https://www.ict.inaf.it/gitlab/odmc/orm_example
- You can clone the project with the git version control system, i.e. with the command:

```
git clone https://www.ict.inaf.it/gitlab/bignamini/orm_project.git
```

- Anyway, to create a Diango project from scratch you can use the following commands

```
django-admin startproject orm_example
cd orm_example
python manage.py startapp imagedb
```

which creates a project folder, named **orm_example**, with additional files and then an application, named **imagedb**, inside the project.
It automatically creates skeleton files needed by a django project and application

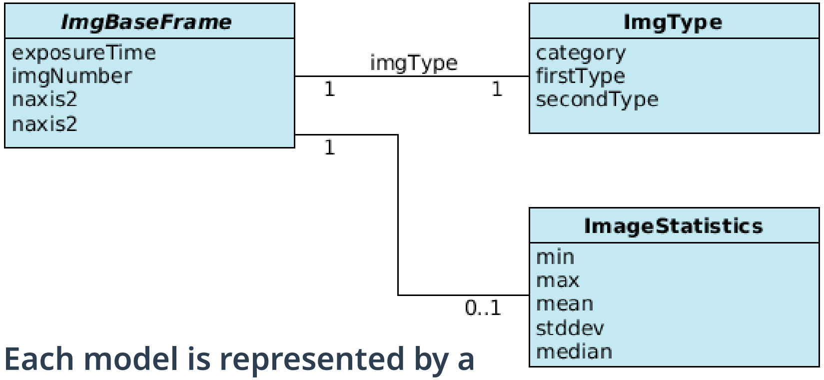
Project structure

```
orm_example/
├── imagedb
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── orm_example
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

- models.py: File containing the app data model
- views.py: Views on the data model classes
- settings.py: Project settings: app list and configuration
- urls.py: Site urls declaration

The Django ORM

- From the data model class to a Django ORM model class



- **Each model is represented by a class that subclasses django.db.models.Model**
- ImageBaseFrame here is **abstract**: no table instantiated
  - That's why we define the stats attribute as a Foreign Key to the ImageStatistics class and not vice versa

```python
from django.db import models

class ImageBaseFrame(models.Model):
  exposureTime = models.FloatField()
  imgNumber = models.PositiveSmallIntegerField()
  naxis1 = models.PositiveIntegerField()
  naxis2 = models.PositiveIntegerField()
  imageType = ImageType()
  stats = models.OneToOneField(
    ImageStatistics,
    models.SET_NULL,
    blank=True,
    null=True,
  )

  class Meta:
    abstract = True
```

Enumerated type

- By enumerated type, or choice, here we mean a type that provides a set of possible values which the attribute is constrained towards
- The Django ORM provides this feature through the `choices` parameter (option) available to all field types

```
logLevel = models.PositiveSmallIntegerField(
           choices=((10, 'DEBUG'),
                     (20, 'INFO'),
                    (30, 'WARNING'),
                     (40, 'ERROR'))
                      )
```

- The choices parameter requires an iterable (e.g., a list or tuple) consisting itself of iterables of exactly two items
- The first element in each tuple is the actual value to be set on the model, and the second element is the human-readable name

Composite fields

- Sometime we would like to define a model class attribute as a **multi-column field** in the same table (i.e. a non-atomic type) instead of creating a 1-to-1 relation (a second table with the attribute columns and a foreign key)
- Many ORM systems provide such feature:
  - JPA: named as **embeddable classes**
  - odb: named as **Composite Value Types**
  - SQLAlchemy: named as **Composite Column Types**
- Django ORM does not provide directly this feature. However there is a package provided by the community, called django-composite-field, which provides an “acceptable” solution
- Composite fields provide an implementation of a “part-of” relationship, i.e. what in the UML class diagram is called **composition**

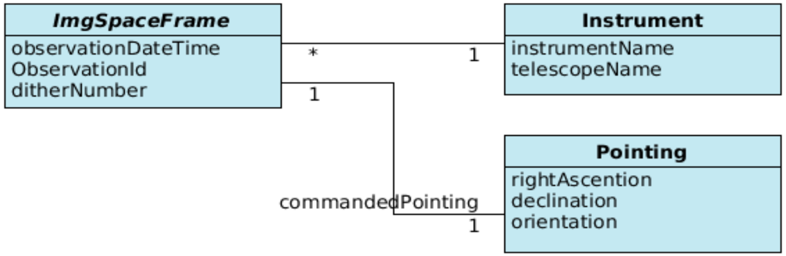
The ImageType class

```
IMAGE_CATEGORY = (
    'SCIENCE',
    'CALIBRATION',
    'SIMULATION')

IMAGE_FIRST_GROUP = (
    'OBJECT',
    'STD',
    'BIAS',
    'DARK',
    'FLAT',
    'LINEARITY',
    'OTHER')


IMAGE_SECOND_GROUP = (
    'SKY',
    'LAMP',
    'DOME',
    'OTHER')
```

```
from composite_field import CompositeField

class ImageType(CompositeField):

    category = models.CharField(
        max_length=20,
        choices=[(d, d) for d in IMAGE_CATEGORY]
    )

    firstType = models.CharField(
        max_length=20,
        choices=[(d,d) for d in IMAGE_FIRST_GROUP]
    )

    secondType = models.CharField(
        max_length=20,
        choices=[(d,d) for d in IMAGE_SECOND_GROUP]
    )
```

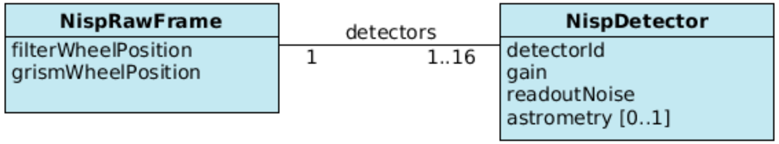
The ImageSpaceFrame class

- The same Instrument is associated to many images, hence here we use a Foreign Key from `ImageSpaceFrame` to `Instrument`
- **If the Instrument instance is deleted, also all images referring to it are automatically deleted (option on_delete set to models.CASCADE in ForeignKey)**




```
class Instrument(models.Model):
  instrumentName = models.CharField(max_length=100)
  telescopeName = models.CharField(max_length=100)


class Pointing(CompositeField):
  rightAscension = models.FloatField()
  declination = models.FloatField()
  orientation = models.FloatField()


class ImageSpaceFrame(ImageBaseFrame):
  observationDateTime = models.DateTimeField()
  observationId = models.PositiveIntegerField()
  ditherNumber = PositiveSmallIntegerField()
  instrument = models.ForeignKey(Instrument,
                                 on_delete=models.CASCADE)
  commandedPointing = Pointing()

  class Meta:
    abstract = True
```

NispDetector

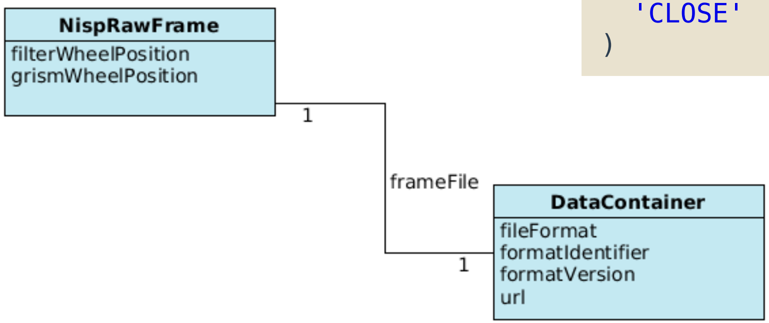


- Many detectors (up to 16) associated to the same raw frame
- Since NispRawFrame is not yet defined, we pass the class name as a string to models.ForeignKey
- But we want to access the detector data using the NispRawFrame class, i.e. the reverse relation.
- This is the purpose of the related_name parameter. For instance we can access the detector data using NispRawFrame.detectors

```
NISP_DETECTOR_ID = (
  '11','12','13','14',
  '21','22','23','24',
  '31','32','33','34',
  '41','42','43','44'
)


class NispDetector(models.Model):
  detectorId = models.CharField(
    max_length=2,
    choices = [(d,d) for d in NISP_DETECTOR_ID]
  )
  gain = models.FloatField()
  readoutNoise = models.FloatField()
  rawFrame = models.ForeignKey('NispRawFrame',
                               related_name='detectors',
                               on_delete=models.CASCADE)
```

NispRawFrame class

- A models.OneToOneField is analogous to models.ForeignKey with the option unique=True but the reverse side of the relation will directly return a single object




```
NISP_FILTER_WHEEL = (
  'Y',
  'J',
  'H',
  'OPEN',
  'CLOSE'
)

NISP_GRISM_WHEEL = (
  'BLUE0',
  'RED0',
  'RED90',
  'RED180'
  'OPEN'
  'CLOSE'
)
```

```
class DataContainer(models.Model):
  fileFormat = models.CharField(
    max_length=10
  )
  formatIdentifier = models.CharField(
    max_length=20
  )
  formatVersion = models.CharField(
    max_length=20
  )
  url = models.URLField()


class NispRawFrame(ImageSpaceFrame):
  filterWheelPosition = models.CharField(
    max_length=10,
    choices = [(d,d) for d in NISP_FILTER_WHEEL]
  )

  grismWheelPosition = models.CharField(
    max_length=10,
    choices = [(d,d) for d in NISP_GRISM_WHEEL]
  )
  frameFile = models.OneToOneField(DataContainer,
                                   on_delete=models.CASCADE)
```

DB Schema creation 1/2

- Once we have defined our data model in **imagedb/models.py** we need Django to create the corresponding DB schema
- First let's check the the project settings includes the imagedb application, i.e. that the file **orm_example/settings.py** contains the the strings highlighted in red in the box on the bottom left
- To do the first migration, i.e. generation of the DB schema, **run the following command**

command

```
python manage.py makemigrations
```

output

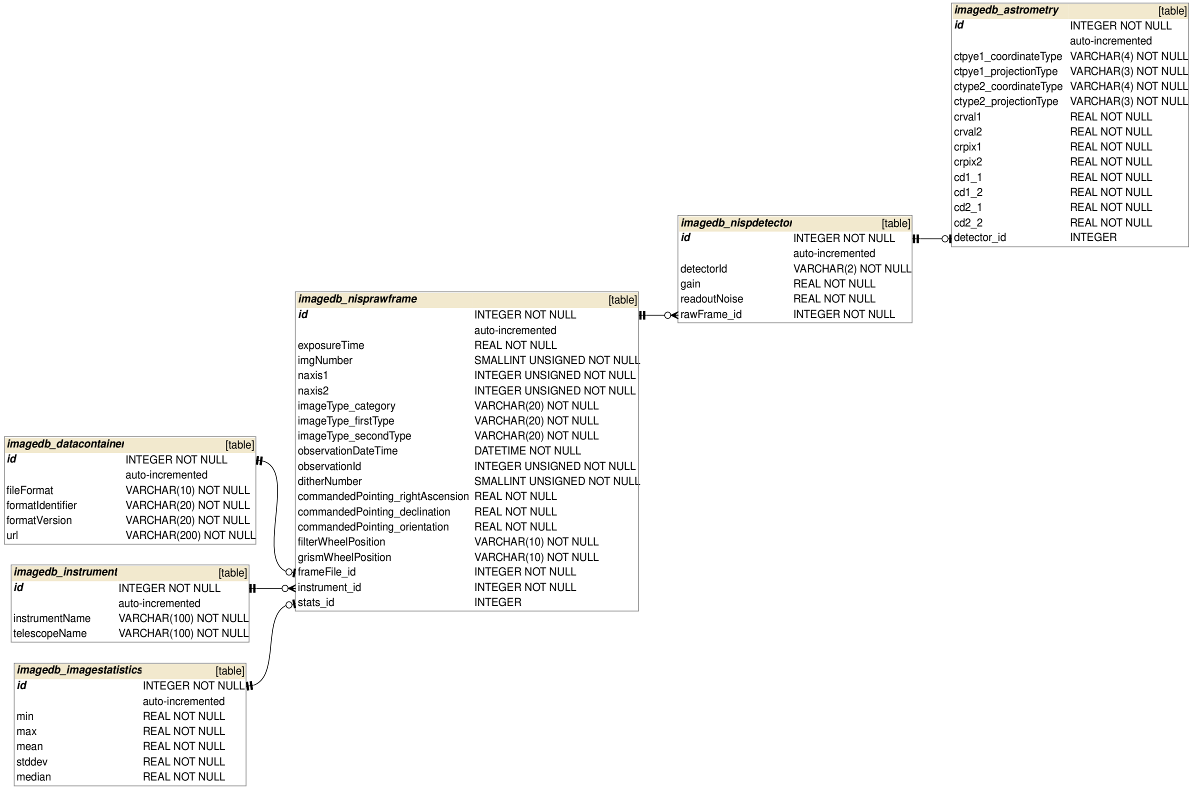
```
Migrations for 'imagedb':
  imagedb/migrations/0001_initial.py
    - Create model Astrometry
    - Create model DataContainer
    - Create model ImageStatistics
    - Create model Instrument
    - Create model NispDetector
    - Create model NispRawFrame
    - Add field rawFrame to nispdetector
    - Add field detector to astrometry
```

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_extensions',
    'imagedb',
    'rest_framework',
    'url_filter',
]
```

Then run the command

```
python manage.py migrate
```

DB Schema creation 2/2



Data insertion and retrieval

- We can now open a python shell and interact with the data model API

```
python manage.py shell
```

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from imagedb.models import Instrument

In [2]: instrument = Instrument(telescopeName='Euclid', instrumentName='VIS')

In [3]: instrument.save()

In [4]: quit
```

- However, for didactic purpose, we can use a Django extension to start a Jupyter notebook. The orm_example project example already provides one notebook. To use it, issue the following command:

```
python manage.py shell_plus --notebook
```

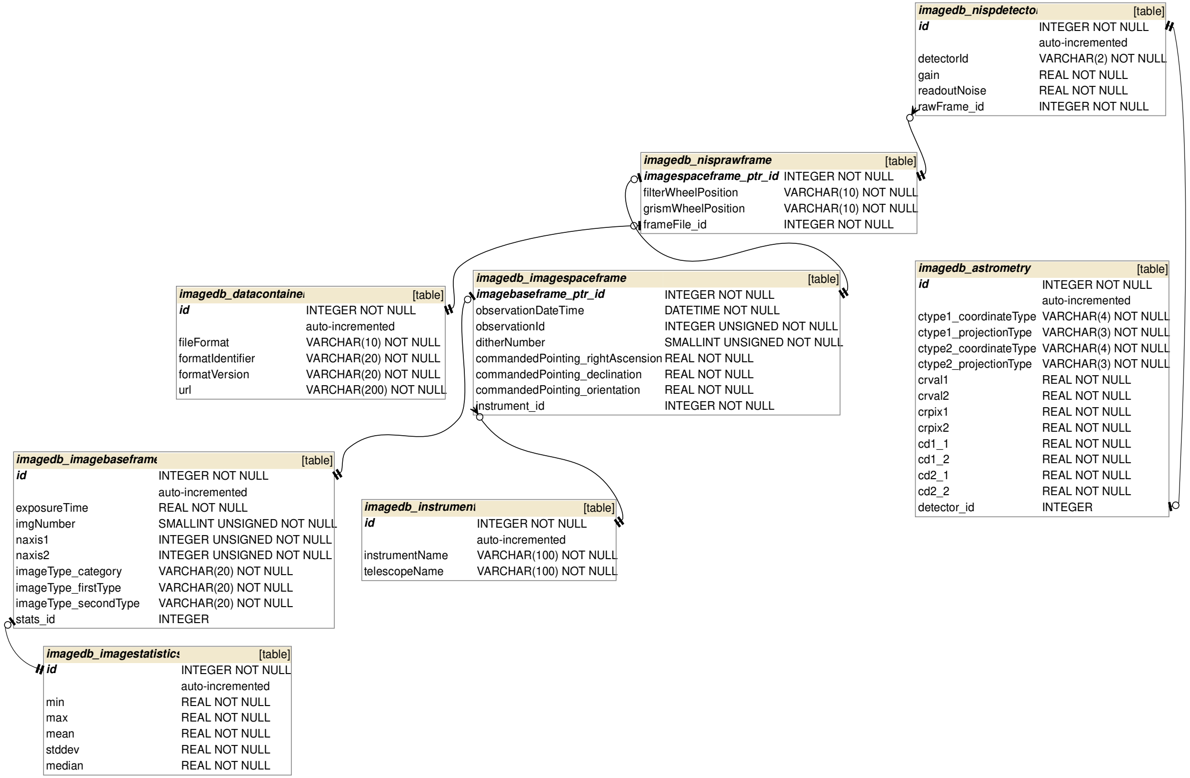
a browser page will be opened. In this page, select the file **imagedb_objects.ipynb** and execute each cell.

Multi-table inheritance 1/3

- With django model abstract base classes, we cannot define foreign keys referencing such base class (since no table is create for abstract classes)
- A solution is the **Multi-table inheritance** of Django models. In this case the abstraction is removed from such base classes and each class in the inheritance hierarchy will have a corresponding table in the DB schema.
- To obtain a multi-table inheritance version of the previous data model, remove the statements

```
class ImageBaseFrame(models.Model):
  ...

  class Meta:
    abstract = True

class ImageSpaceFrame(ImageBaseFrame):
  ...

  class Meta:
    abstract = True
```

Multi-table inheritance 2/3



Multi-table inheritance 3/3

- **Each model corresponds to its own database table and can be queried and created individually**
- **The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField)**
- **With the multi-table inheritance, all fields of ImageBaseFrame will still be available also in ImageSpaceFrame and NispRawFrame**
- If we have an ImageBaseFrame instance that is also an ImageSpaceFrame instance, we can get from ImageBaseFrame object to ImageSpaceFrame object by using the lower-case version of the model name

```
from imagedb.models import ImageBaseFrame

obj = ImageBaseFrame.objects.get(pk=2)
obj.imagespaceframe.nisprawframe
```

```
<NispRawFrame: NispRawFrame object (2)>
```

Serializing Django objects

- **Django's serialization framework provides a mechanism for "translating" Django models into other formats.**
- **Usually these other formats will be text-based and used for sending Django data over a wire, but it's possible for a serializer to handle any format (text-based or not).**
- **Django supports a number of serialization formats, including XML and JSON.**

```
from django.core import serializers

serializers.serialize('json',NispRawFrame.objects.filter(observationId=53877,
                     filterWheelPosition='Y').order_by('ditherNumber'))
```

- **The Django serialize function requires, as one of the inputs, a QuerySet**
- However, the **Django REST framework**, external to the Django framework, provides a more flexible serialization mechanism

The Django REST serializers

- In particular, the Django REST framework provides a **ModelSerializer** class which can be a useful shortcut for creating serializers that deal with model instances and querysets
- See ‘imagedb/serializers.py’ to check some examples

```
from rest_framework import serializers
from composite_field.rest_framework_support import CompositeFieldSerializer

...

class NispRawFrameSerializer(serializers.ModelSerializer):
  detectors = NispDetectorSerializer(many = True, read_only = True)
  commandedPointing = CompositeFieldSerializer()
  imageType = CompositeFieldSerializer()

  class Meta:
    model = NispRawFrame
    exclude = [f.name for g in NispRawFrame._meta.get_fields()
                if hasattr(g, 'subfields')
                for f in g.subfields.values()]
    depth = 2
```

The Django REST framework

- **We need an Application Programming Interface (API) that let us perform CRUD operations on the database without directly connecting to the database**
- **A REST (Representational State Transfer) API provides such operations through HTTP methods:**
  - GET, to request to a server a specific dataset
  - POST, to create a new data object in the database
  - PUT, to update an existing object in the database or create it if it does not exist
  - DELETE, to request the removal of a given data object
- **Such methods can be applied to a specific set of endpoints (URLs) provided by our API**
- **The Django REST framework provides software tools to build a REST API on top of our models**

Django REST framework ViewSets

- The actions provided by the **ModelViewSet** class are .list(), .retrieve(), .create(), .update(), .partial_update(), and .destroy() of instances of a specific model we have defined
- The **ReadOnlyModelViewSet** only provides the 'read-only' actions, .list() and .retrieve()
  - In practice it returns a list of instances of a specific model or it retrieves a single instance by its primary key value
- In our orm_example projects, we have few examples in **imagedb/views.py**

```
from rest_framework import viewsets
from imagedb.serializers import NispRawFrameSerializer

class NispRawFrameViewSet(viewsets.ReadOnlyModelViewSet):
  queryset = NispRawFrame.objects.all()
  serializer_class = NispRawFrameSerializer
```

- More advanced filtering capabilities can be added with additional parameters: https://www.django-rest-framework.org/api-guide/filtering/

URLs

- Once we have defined viewsets on our models, we have to create endpoints (urls) to access those views
- The Django REST framework provides the so called **routers**, which generate automatically url patterns based on the views we have defined
- An example is found in **imagedb/urls.py**

```
from django.conf.urls import url, include
from rest_framework.routers import DefaultRouter

from imagedb import views

router = DefaultRouter()
router.register(r'nisprawframes', views.NispRawFrameViewSet)

urlpatterns = [
  url(r'^', include(router.urls))
]
```

will generate automatically the following url patterns:

/nisprawframes/ : **it will return, in json format, all the NispRawFrame objects in the database**

/nisprawframes/[pk]/ : **it will return only the NispRawFrame object with primary key pk**

Starting the Django development server

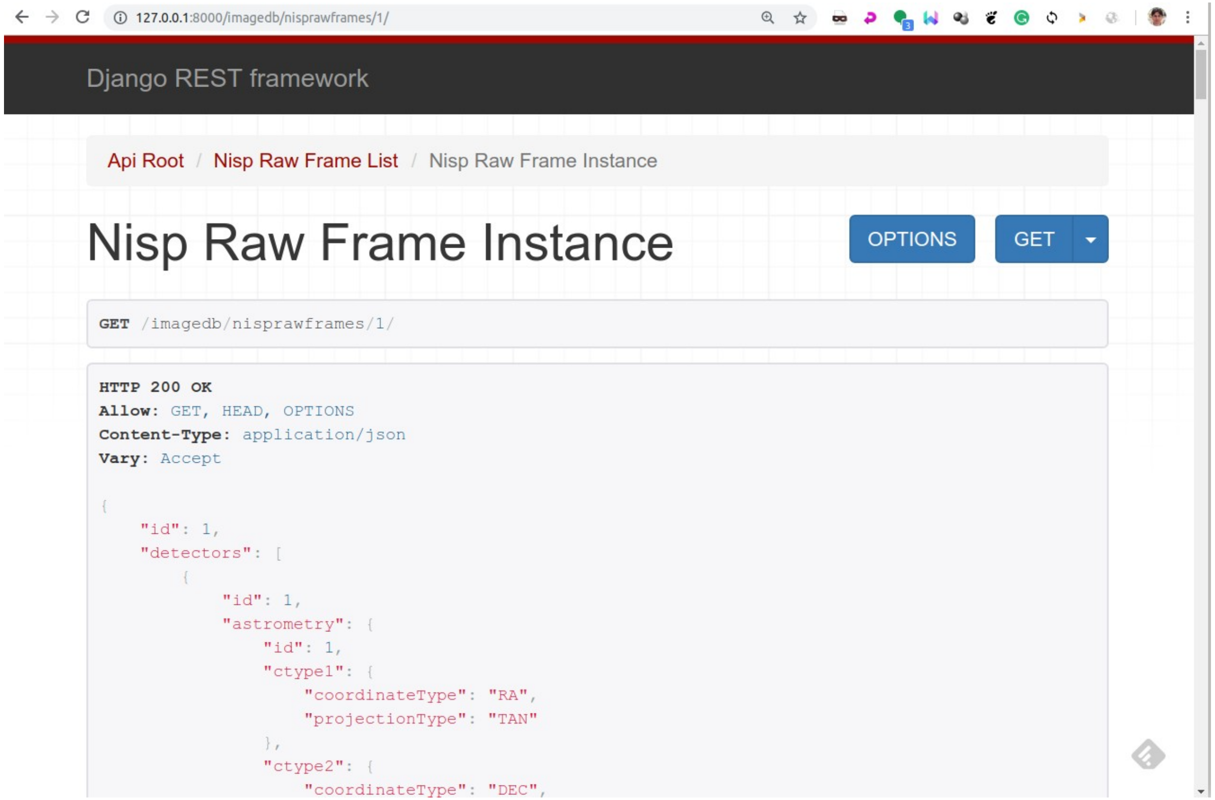
- In order to test the REST API, you can start the Django server with the following command

```
python manage.py runserver
```

```
Performing system checks...

System check identified no issues (0 silenced).
October 15, 2018 - 21:30:57
Django version 2.1.1, using settings
'orm_example.settings'
Starting development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

- Now with the browser you can open the following link: http://127.0.0.1:8000/imagedb/nisprawframes/1/

The browsable REST API



More advanced filtering criteria

- In order to use more advanced filtering criteria through the REST API, rather then just the primary key, in the orm_example project we have added the **django-url-filter** (https://github.com/miki725/django-url-filter)
- With this filter, we can specify filtering condition directly in the url, e.g. :

```
http://127.0.0.1:8000/imagedb/nisprawframes/?observationId__in=53877,54349&filterWheelPosition=Y
```

References

- https://www.yworks.com/products/yed
- https://www.lucidchart.com
- https://plantuml.com
- https://www.djangoproject.com
- https://www.youtube.com/watch?v=UI6lqHOVHic
- https://www.uml-diagrams.org
- https://www.guru99.com/uml-diagrams.html
- Blaha, Michael. (2013). UML Database Modeling Workbook
- https://www.djangoproject.com
- https://www.sqlalchemy.org