BIRD
Banks' Integrated Reporting Dictionary

Version 1.4

August 2023

BIRD Logical Data Model design principles

Content

0. Version Control 3

1. Background information 3

2. Principles 4

Principle 1: We separate our concerns 4

Principle 2: We model that for which we have a requirement 4

Principle 3: We model the least granular option that covers all requirements 5

Principle 4: We use business language in the LDM 5

Principle 5: We do not use abbreviations 5

Principle 6: We are as explicit as possible 6

Principle 7: The model should satisfy the third normal form 6

Principle 8: We use subtyping to help us being as explicit as possible 6

Principle 10: we model roles of entities to make the model more explicit 7

Principle 11: We use attributive entity types for specific traits 8

Principle 12: We use generalizations where objects share common traits, but have different primary keys 10

Note on forward engineering generalization 11

Principle 13: Use of associative entities to model many-to-many relationships 11

Principle 14: Relationships between entity types are always as explicit as possible by connecting sub types or using roles 13

Principle 15: Numbers are not numeric 13

Principle 16: We use indicators instead of Boolean values 13

Principle 17: We draw the diagram top down, left to right. 13

Principle 18 Attributes are listed from primary key to discriminator 14

3. Annex I 14

3.1 Quick Reference on writing definitions 14

3.2	**Is it a Thing, a Thing of a Thing, or a Thing pointing to another Thing?**	14
3.3	**In General**	15
3.4	**Defining entities, not including associative entities**	15
3.5	**Associative**	16
3.6	**Attribute**	16
3.7	**Relationship**	17
3.8	**Legend**	18
4.	**Annex II**	**18**
4.1	**Declarative versus imperative data validation rules**	18

0. Version Control

Version	Date	Comments
1.0	07/10/2020	Initial draft.
1.1	12/08/2021	Editorial amendments, amendments of examples to reflect the current situation in the LDM, incorporation of suggestions by members of the Work stream on Data Modelling. Version released internally only
1.2	10/10/2021	Incorporation of comments and suggestions for improvement provided by members of the Work stream on testing (WS T). Version released internally only
1.3	16/12/2022	Incorporation of comments and suggestions for improvement provided by members of the Work Stream on Prototyping (formerly known as Work Stream on Testing) and of the temporary BIRD subgroup on logical data model and input layer (LDM/IL) review.
1.4	30/08/2023	Added principle for the order of attributes in the entity type.

1. Background information

The ECB project to harmonize reporting, Banks' Integrated Reporting Dictionary, or BIRD for short, aims at capturing all reporting requirements and describing the common ground between those requirements. To be able to specify exactly what those data requirements look like and how they fit together, a logical data model (LDM) is designed using the notation and technique of the entity relationship model (ERM). It is the responsibility of the Work Stream on Data Modelling (WS DM) to design and maintain it. This logical data model describes all elements needed for the reports. Please note that this LDM also comprises derived information and may therefore also be labelled Enriched-Logical Data Model (ELDM). For the sake of simplicity, we will refer to it as LDM in this document. We would also like to stress the point that the LDM describes what is required and not how it should be implemented, it is a logical data model, not an implementation model.

The logical data model is designed according to the principles that are laid-out in this document.

Please be aware that we use so-called crow's foot notation[1] (or also called Information Engineering (IE) notation) when drawing the model.

2. Principles

The main guiding principle is that we separate our concerns. And we separate them into various layers of representation of data. The commonly accepted levels are the semantical level, the logical level, the technical level, and the physical implementation level. Further details about the different levels may be found in the paper Heading for harmonization of data collection[2]. Each level deals with its own concerns. In short, the semantical level deals with the meaning and definition of the data. It structures the policies and regulations so that they are consistent with each other. The analysis of this semantic layer is mainly the work of the various Expert Groups within BIRD. The logical level takes the semantics and written constraints and creates data structures in the logical data model. These structures ensure the correct form of the data. The LDM can do this, because of its underlying relational algebra, which relies heavily on set theory and predicate logic. The technical level deals with concerns surrounding transport and persistence. It deals with operational aspects as timeliness and representation. The physical implementation deals with the technology (RDBMS / Hadoop / …) and concerns in that area.[3]

Please note that the presented principles are principles rather than strict rules. Therefore, the modelling approaches applied in the LDM may deviate from these principles if there is good reason to do so[4].

Principle 1: We separate our concerns

The logical data model deals with concerns of integrity, consistency, and correctness of the data structures. It specifically does not create definitions or makes up names. And likewise, it does not concern itself with implementation details. It "only" describes how the data relates to each other and how they should be structured in a consistent way.

Principle 2: We model that for which we have a requirement

To be able to separate our concerns correctly and not make up semantics, we mainly[5] model that for which there is a requirement. In the context of the BIRD LDM, this entails modelling mainly those requirements

[1] Further information about crow's foot notation may be found online, for example https://www.vertabelo.com/blog/crow-s-foot-notation/

[2] See https://www.bis.org/ifc/events/isi_wsc_62/sts442_paper3.pdf

[3] For an overview of all levels of representation, the concerns they address and the modelling techniques to apply for specific situations, please have a look at the poster of i-refact: https://www.i-refact.com/wp-content/uploads/2018/09/DataModelMatrixPoster0.1j.pdf

[4] See also https://en.wikipedia.org/wiki/Leaky_abstraction

[5] Some additional modelling features can be defined in order to complement other fields or be used to define validation rules

that follow from the reports under consideration. On a more advanced stage of the project, also ad-hoc reporting requests might be considered. The data requirements from these reports, as described by the Expert Groups, are integrated in the already existing LDM. Where necessary the LDM is refactored accordingly in order to ensure integrity, consistency and correctness of the data (structures).

Principle 3: We model the least granular option that covers all requirements

This principle follows directly from Principle 2: We model that for which we have a requirement. The logical data model for BIRD finds the common ground in all the reports covered by the BIRD documentation and finds the highest level of aggregation of the data that fits all reports. In this respect it is very similar to finding the highest common denominator of two or more numbers. This is always the equal or smaller than the lowest of those numbers. The least granular data model option that covers all reports is at least as detailed as the most detailed report.

The flipside of this is that we do not model even more granular options, even when they do exist. For example, there is a requirement for modelling security positions. The logical data model distinguishes between long and short positions held in specific securities. These positions are calculated based on all individual transactions involving the specific securities. Since there is a requirement to report the position, it is part of the model. There is no requirement (yet) to deal with the individual transactions, so those are not part of the logical data model.

Principle 4: We use business language in the LDM

The logical data model does not invent data structures. It only reveals them from the semantical structures that are already defined. Only in this manner can the logical data model be the bridge between business requirements and technical implementations of these requirements. The best way to guarantee that the logical data model is understandable to the business side is to use their words.

However, since the logical data model is a harmonized layer used for a several reports, report-specific terms or classification should be limited as much as possible. The information contained in the logical data model should converge to their original business meaning.

Principle 5: We do not use abbreviations

To correctly use business language, and to be able to make sense to the widest possible group of people that will work with the logical data model, we refrain from using abbreviations in the names of the different terminologies involved. Abbreviations are almost always used as business shorthand and degenerates into jargon. When one is not exposed (anymore) to that jargon, an abbreviation might become meaningless. Thus, we do not use abbreviations, but we write the abbreviation out in full. So Lower of Cost or Market instead of LOCOM; reporting agent instead of RA, reporting member state instead of RMS.

Some abbreviations are so common that they have become a word in the general language – think of ECB and GAAP. It is ok to use those, but they should be used sparingly.

Principle 6: We are as explicit as possible

Since the logical data model for BIRD concerns itself solely with correctness of structure and integrity of data, the Work Stream on Data Modelling decided to be as explicit as possible with its modelling. This allows for the highest number of requirements to be captured in a declarative manner in the LDM[6]. By being as explicit as possible in LDM, data structures are valid by default and therefore there is only limited need for additional validation rules.

Principle 7: The model should satisfy the third normal form

In order to be as explicit as possible and reduce redundancy each attribute contains only one type of information and so-called functional dependencies[7], i.e. dependencies between attributes, should be removed. For example, the name and the address of a company should not be included in the same attribute but separated in two specific attributes.

Principle 8: We use subtyping to help us being as explicit as possible

The logical data model for BIRD makes copious use of subtypes. Subtypes allow us to define distinct subsets of all occurrences of the super entity type and describe specific attributes to them. In this manner, the attributes are described as explicit as possible, since they are described at the level where they belong. Were they described at a higher level, then the attribute had to be optional because it is not valid for all occurrences of the super type.

For example, in the context of Securities, the Legal final maturity date is only applicable to Debt securities but not to Equity and fund securities. By creating dedicated subtypes, i.e. Debt security and Equity and fund security, the attribute Legal final maturity date can be declared mandatory. Otherwise, if it would have been located on the level of Security, the attribute had to be declared optional and a validation rule had to be created to ensure that this attribute is only populated for Debt securities but not for Equity and fund securities. This way we integrate the business rule, specifying that: if a Security is a Debt security the Legal final maturity date must be provided.

To illustrate the above stated point, please find here the underlying model design:

---

[6] See **Declarative versus imperative data validation rules**

[7] Third normal form: https://en.wikipedia.org/wiki/Third_normal_form

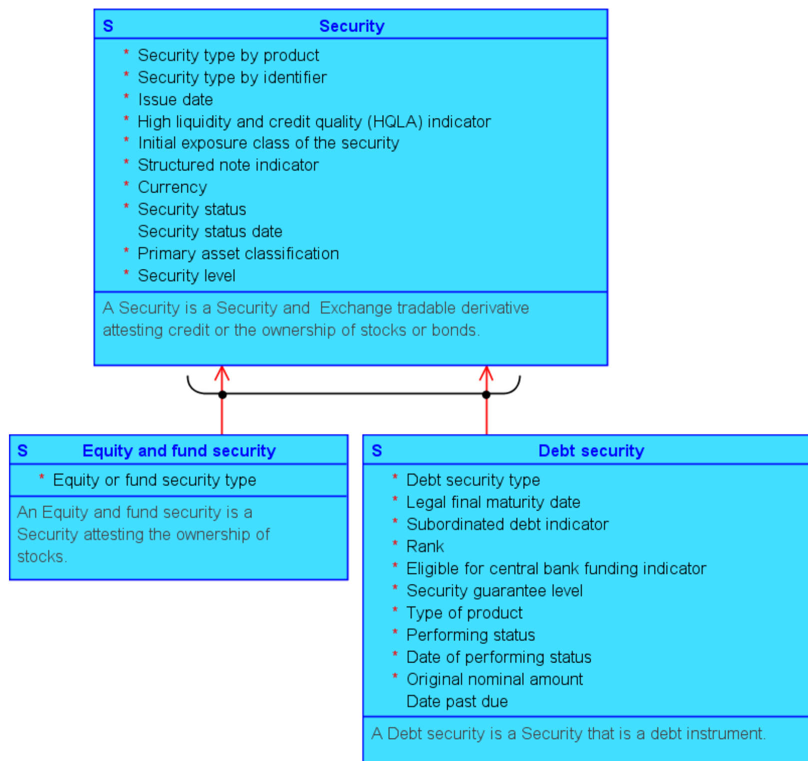

Figure 1: Equity and fund security and Debt security as subtypes of Security having different attributes

Disjoint subtyping, that is subtyping into different sets of subtypes, on the same level is allowed and sometimes necessary but it should be evaluated if the model becomes unnecessarily complex because of the application of this modelling method.

Principle 10: we model roles of entities to make the model more explicit[8]

Depending on the context, an entity[9] may act in different roles. For example, a party might be relevant to the financial institution because it is the issuer of a security, or because it is the debtor of a loan. The following picture indicates the application of the role concept in the model, i.e. a *Party* acts in many roles, one of them might be *Creditor*, another one Loan d*ebtor*.

[8] We used to have a Principle 9 but it was agreed in the Work Stream on Data Modelling to remove it. We kept the numbering of the principles the same, because we are using the numbers in other documents. If we change them, all the references in other documents would need to change.

[9] The term entity here represents a real-life business object under the scope of the model

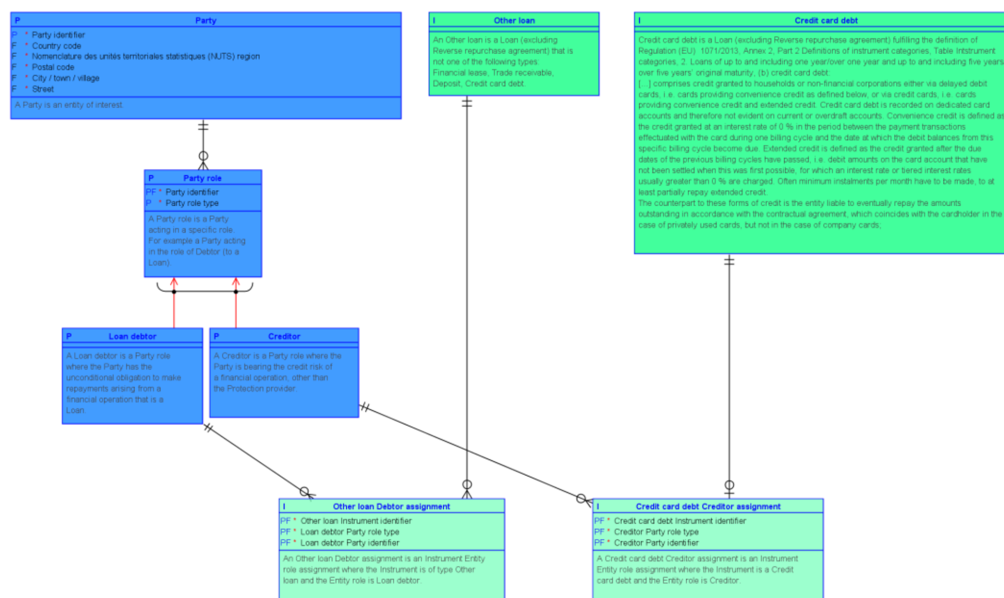

Figure 2: Party acts in roles of Creditor, Loan debtor

By using specific roles in certain connections, the model explicitly specifies that only entities acting in these roles are applicable to a certain connection. For example, an *Other loan* has one-or-many Loan d*ebtors* which is reflected in the entity type *Other loan Debtor assignment* which has a relationship with the *Loan debtor* entity type and therefore only *Parties* acting in the *Party role Loan debtor* are allowed to be *Debtors* of such *Loans*. Additionally, this approach allows us to use business language (in line with Principle 4: We use business language in the LDM), so we use *Investor* for *Security positions*, instead of using more generic terms like *Creditor*.

An entity can take several roles at the same time, consequently the primary key of the Role Entity always contains the role type.

Principle 11: We use attributive entity types for specific traits

Attributive entity types are a model construct that is used in two distinct ways in the BIRD LDM. Both share the same characteristics in that the attributive entity type has as primary key, the primary key of the entity type it belongs to (e.g. *Instrument risk data* is an attributive entity type connected to the *Instrument* entity).

The first way is where it is clear that an attribute can only belong to a certain entity type, but that somehow it can occur multiple times at the same point in time. For example, a *Party* can have multiple *Enterprise sizes (calculated)* from multiple previous years, that are required input for the enterprise size calculation of the current reporting period. Therefore, the entity type *Party previous period data* has a many-to-one connection with *Party*. In this case, the primary key of the entity type it belongs to, is complemented with another attribute that enumerates the values the attribute can have. The following picture illustrates this situation for the entity type *Party* which can have multiple *Party previous period data* values. Please note

that the primary key of the entity type *Party previous period data* is composed of the primary key of the entity type *Party* (which is *Party identifier*) and an additional attribute (in this case. *Year*) which enumerates the different values.

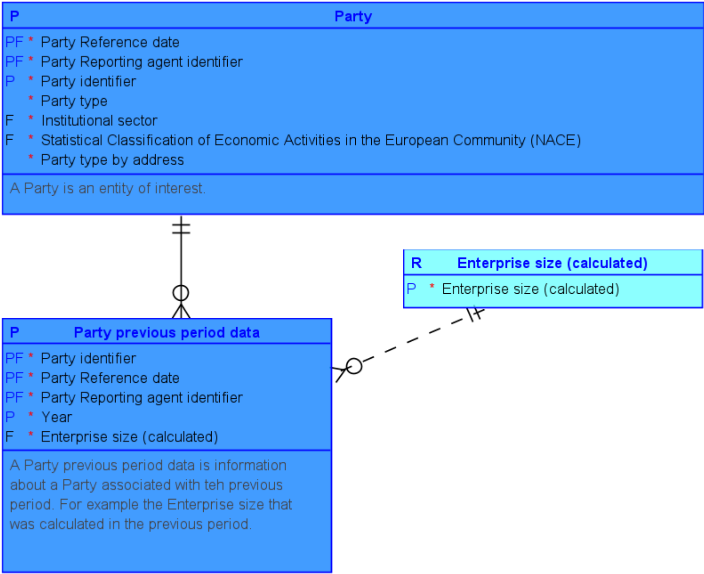

Figure 3: Party has one-or-more enterprise sizes from previous period

The second possibility is where the attributive entity type is used as a de-facto subtype, where we do split the set of occurrences that is the parent entity type, into one specific subset, but where there is no need to define the inverse of the set. One example of this is the entity type *Party* which has an attributive entity type *Party risk data* where attributes like *Performing status, Default status* and *Date of default status* are defined. The primary key of *Party risk data* is exactly the same as the one of *Party*, so, every *Party risk data* is a *Party*. However, the inverse is not always true, there exist *Parties* in the scope of the LDM that do not fulfill the criteria of having *Party risk data*. The following picture illustrates this situation in the LDM.

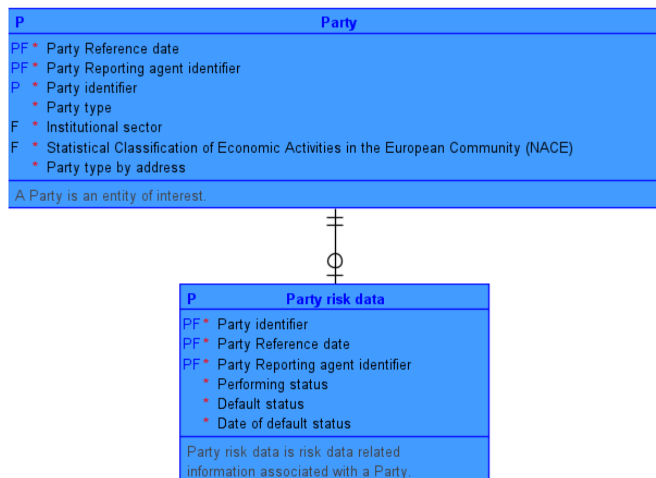

Figure 4: implicit subtyping of Party into Party risk data

It is important to highlight that this de-facto subtyping, as illustrated above using the *Party risk data* example, implies the drawback that the de-facto subtype is not identifiable via discriminators, i.e., there exists no attribute which allows to select only this kind of subtype. Given the above illustrated example the consequence is that *Party* **without** *risk data* cannot be identified directly (because of the absence of a discriminator) but its existence has to be derived from the absence of *Party risk data*.

Principle 12: We use generalizations where objects share common traits, but have different primary keys

The generalization pattern in the BIRD LDM is one of the more advanced patterns that is applied in the model. It occurs when in the semantic layer there is a hypernym – hyponym relation between two terms and when the entity types that result from the hyponyms turn out to have different primary keys. Simply speaking, we can use the generalisation pattern if there exists a feature that is applicable for two entity types which have different primary keys, e.g., the accounting classification is applicable for instruments and security positions although they are identified by different primary keys. In the logical data model, this construct helps us to define abstract concepts relatable to concrete concepts (for example *Non-financial asset and non-financial liability* with the carrying amount attribute as a generalization of *Non-financial asset* and of *Non-financial liability*).

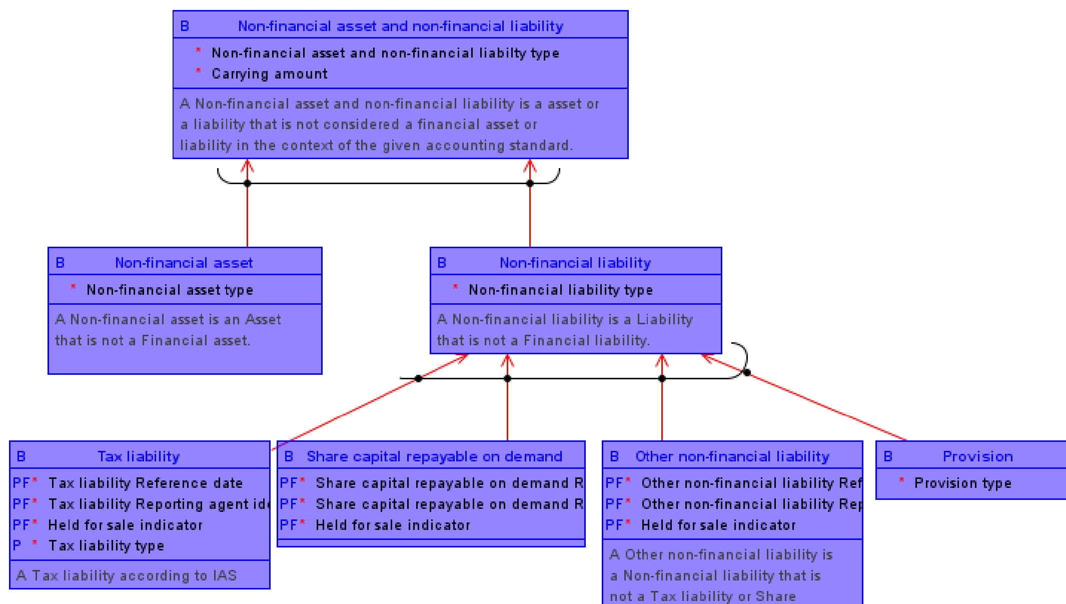

Figure 5: Non-financial asset and non-financial liability as a generalisation sharing the carrying amount in all subtypes, where the subtypes have distinct primary keys. This example is a partial extract of the full generalisation in the LDM.

Note on forward engineering generalization

When there is a super type with two subtypes, the normal situation is that they all have the same primary key. So, if there is a relationship type from the super type to a separate entity type, the foreign key will just be pointing to the primary key of the super type. With generalizations, this works differently. In the technical implementation, the foreign key that points to the generic super type must be implemented as one foreign key for each specific subtype instead. This is because the primary key is defined on the subtype level, which means that the foreign key can only be implemented on that level. The ease of modelling and declaring that generalisations bring in the logical data model is counteracted by the more difficult implementation.

Principle 13: Use of associative entities to model many-to-many relationships

Relationships between two entity types (e.g. *Instrument* and *Party*) can be either modelled using a direct relationship or an associative entity.

Relationships that are of type one-to-one or one-to-many can be modelled as direct relationships. An example of such a direct relationship is the relationship between an *Organisation* and associated *Organisational unit*, where an *Organisation* comprises / includes (optional) one-or-many *Organisational unit(s)*, as illustrated in the following picture.

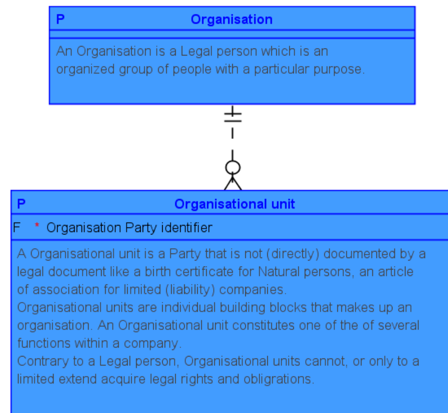

Figure 6: Organisation has one-or-many (optional) Organisational units

Relationships of type many-to-many are modelled using associative entities, for example the relationship between a *Loan* and its *Loan debtor(s)*, where a *Loan* has one-or-many Loan d*ebtor(s)* while a *Loan debtor* is obliged to pay (the *Outstanding nominal amount*) of one-or-many *Instrument(s)*, as illustrated in the following picture.

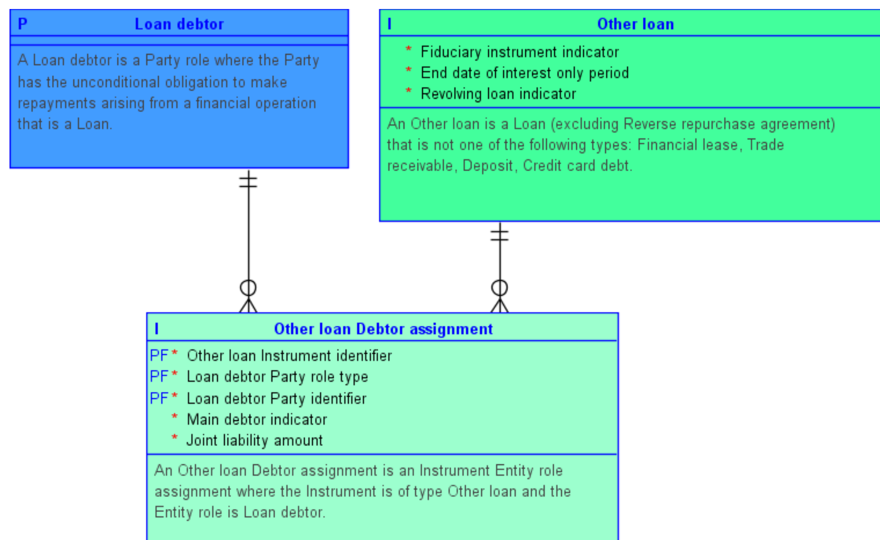

Figure 7: Debtor and Instrument's many-to-many relationship modelled using associative entity Instrument debtor assignment

In such an associative entity the primary key is composed of (at least) the primary keys of the entities it connects. In the above illustrated picture this is the primary key of the *Instrument* (i.e. *Other loan Instrument identifier*) and the primary key of the Loan d*ebtor* party role (i.e. *Loan debtor Party identifier*, *Loan debtor Role type*). Please note that such associative entities are explicit in the sense that they link only specific types, in this case the *Loan* with the *Loan debtor*. If we want to establish another many-to-many association

between *Instrument* and *Party*, e.g. the *Servicer* of an *Instrument*, we create another associative entity for this purpose.

Principle 14: Relationships between entity types are always as explicit as possible by connecting sub types or using roles

In order to ensure the highest level of integrity of the data (structures) it is important that relationships are as explicit as possible. This implies that relationships are either created between the sub types of entity types where applicable, or that relationships are established using roles. Examples for the application of this principle are the associations between *Loans* and their *Debtors*, *Creditors* or *Servicers* (where roles are used) or the relationship between *Organisation* and *Organisational unit*.

Principle 15: Numbers are not numeric

In the data base world, we are prone to store numbers in a numeric field. It just *feels* right. They both consist of digits and integer data type is made for storing digits. However, most values that are called numbers, are not numeric. Things like street number, telephone number, tax identification number, et cetera are prime examples of this. With street numbers, there might be an addition, with phone numbers, you might want to have a '+' or a '0' as the first character, and the tax authority might introduce hyphens or letters to the tax identification number.

Numeric storage only ever makes sense if you can carry out meaningful calculations to the values. Dividing a telephone number by 3 does not give a meaningful result. Likewise adding to tax identification numbers together is not useful. That is why we store "numbers" as text and only numeric values where meaningful calculations are applicable as numbers.

Principle 16: We use indicators instead of Boolean values

Instead of using Boolean values True and False we use indicators. For example, the Attribute *Subordinated debt indicator* has allowed values *Subordinated debt* and *Non-subordinated debt*. This approach ensures that the LDM is flexible regarding possible changes in the allowed values in the future. Additionally, this approach ensures the utility of business language in the model, for example in the LDM we would speak about *Subordinated debt* instead of *Subordinated debt indicator* = "True". The value "True" itself does not hold any business meaning and needs to be mapped to the correct interpretation while *Subordinated debt* has a business meaning.

Principle 17: We draw the diagram top down, left to right.

We draw the diagram of the logical data model in a specific manner. Fundamentally there is no such thing as a direction in a data model diagram. Everything is equally valid at the same time and there is no flow to

describe. However, we have found that presenting the logical data model in a top-down, left-to-right fashion helps in quickly understanding the structure of the model.

These are the steps we use:

- First, identify the *kernel* entity (e.g. *Contract*, *Instrument* etc.) types and place them at the top next to each other from left to right. Kernel entity types form the heart of the model and everything follows from them. They have relationships and subtyping going from them, but they are themselves not a subtype, nor do they have a foreign key.
- Second, subtypes go below and every subtype on the same level is placed next to each other. This visually indicates that they are on the same level.
- Third, the entity types that are on the foreign key side of a relationship type are placed below the entity types where the one-side of the relationship type is.
- Fourth, a relationship type is drawn in such a manner that the “one” side is **always** coming from the bottom part of the entity type and the “many” side is **always** entering at the top. This also **always** applies to subtyping constructs, they also go from the bottom of the super type to the top of the subtype.

Principle 18 Attributes are listed from primary key to discriminator

The order of the attributes within the entity type is from the primary key to the discriminator.

- First, we list the attributes that are part of the primary key and that are inherited from a relationship type.
- Second, we list the attributes that are part of the primary key that are not part of a foreign key.
- Thirdly, we list all other attributes, except those other attributes that are discriminator attributes.
- Last, the discriminator attributes that are not part of the primary key are listed.

3. Annex I

3.1 Quick Reference on writing definitions

Trying to write a good definition is not easy. By following the structure in this template, the "only" thing that remains is making clear what the term means.

3.2 Is it a Thing, a Thing of a Thing, or a Thing pointing to another Thing?

Does your business term describe itself, does it help describe another term, or does it help to associate two terms?

	Thing	**Thing of a Thing**	**Thing pointing to another Thing**

Also known as	• Business Term • Entity • **Entity Type** • Table • Object	• Business Term • **Attribute** • Attribute Type • Property • Field • Column	• Business Term • Relationship • **Relationship Type** • Foreign Key • Association

3.3 In General

A definition cannot refer to itself, nor are circular references in definitions allowed. A definition states what a thing is, not what it does. It's not a description, it's a definition.

In General

- States what the term is, not what it does
- Names in singular
- No homonyms
- No synonyms
- No abbreviations in the name or description
- No circular references

3.4 Defining entities, not including associative entities

Entities are also known as entity types or objects or just plain things, that we as business keep track of, or as business terms.

Entity Definition

Describe what the entity is, not how it is used or calculated.

Example	A *Patient* is an *Individual*, who is under treatment of a *Physician.* A *Patient* will not be registered as such when he or she is not insured.
Definition template	A <entity name> is a <encompassing term>, <characteristic properties>. [<recording condition>]
<entity name>	Entity name is a commonly used business term that is to be defined using singular nouns.
<encompassing term>	Encompassing term is a more common term whose meaning encompasses the meaning of the entity type name. This encompassing term can be an already defined business term, in which case the entity it represents must

	be considered a sub-type of the encompassing term, or the term is a commonly known term.
<characteristic properties>	Characteristic properties are those properties that make the encompassing term the entity name.
<recording condition>	A restriction or enhancement, indicating when an occurrence of this entity will be recorded in the system. This is only allowed, and mandatory, when not all occurrences of this entity are added to the system.

3.5 Associative

Describing the associative is only applicable when the associative term is not (commonly) known within the organization. (A known business term would result in an entity, so use that template for the definition.)

Associative Definition
Describe why the one part links to the other part.

Example	A *Physician/Specialty* is the combination of *Physician* and *Specialty* that indicates which *Physician* has which *Specialty*, when *Physician* has indicated they want to use that *Specialty*.
Definition template	<Associative Name> is the combination of <entity name 1> and <entity name 2> that indicates which <entity name 1> <verb phrase> which <entity name 2>[, when <conditions>].
<associative Name>	<entity name 1>/<entity name 2> (only when there is no business term covering this combination)
<verb phrase>	Verb phrase describes the association between <entity name 1> and <entity name 2>.
<conditions>	Conditions indicate when an occurrence of this associative needs to be recorded in the system.

3.6 Attribute

An attribute is also known as a property, thing of a thing, field, or column.

Attribute Definition
Describe the role of the attribute for the entity. *What does it do for the entity?*

<table>
<tr><td>Example</td><td>Nurse Telephone Number is a number,
of a telephone connection where the Nurse can be reached at home, as stated by the Nurse.</td></tr>
<tr><td>Definition template</td><td><attribute name> is a <standard descriptor>, <remainder>, the <entity name>, <remainder>, [as {determined|described|stated} {by|in} the <external source>].</td></tr>
<tr><td><attribute name></td><td>The name of the attribute. Names should be singular.</td></tr>
<tr><td><standard descriptor></td><td>• amount <preposition> <unit of measure>
• date
• timestamp
• name
• number
• percentage
• permillage
• ratio
• free format text.</td></tr>
<tr><td><unit of measure></td><td>Unit of measure is either a standard like dozen, kilogram, thousand, or pointing to another attribute, e.g. for the currency.</td></tr>
<tr><td><entity name></td><td>Entity name is the name of the entity where the attribute belongs to.</td></tr>
<tr><td><remainder></td><td>Remainder is anything else needed to make the role of the attribute for the entity clear.</td></tr>
<tr><td><external source></td><td>An organization outside the organization or a department inside of the organization that sets the values. Also known as the data owner.</td></tr>
</table>

3.7 Relationship

Other names for relationship are relationship type, associative or foreign key, or thing pointing to another thing.

Definition Relationship

Describe on the "many" side why that specific instance of the "one" side is chosen.

Why is the nurse working at the intensive care department and not at the geriatric department.

Name	Nurse helps Physician

Name template	<subject> <verb phrase> <object>
Example	When a Nurse is scheduled to help a Physician and having to carry out the standard nursing tasks in order to help the Physician, then it is true that Nurse helps Physician. To help means here to act in the way as prescribed in the standard operating procedures for a Nurse.
Definition template	When <conditions>, Then it is true that <subject> <verb phrase> <object>. [<verb phrase> means here <meaning>]
<subject>	Subject is the entity that is acting as the subject in the relationship.
<object>	Object is the entity that is acting as the object in the relationship. This is most likely the "many" side of the relationship.
<verb phrase>	Verb phrase describes, with a part of a sentence that must contain a verb, how the <subject> acts for the <object>.
<conditions>	Conditions indicate why one specific occurrence of <subject> is chosen to link with the <object> above another occurrence of <subject>.

3.8 Legend

< >	variable. A variable is always explained, except when the meaning is immediately clear.
[ ]	optional
{ }	list of options
( )	clarification

4. Annex II

4.1 Declarative versus imperative data validation rules

In a logical data model, the constraints are declared by the structure of the model. These constraints can be implemented in a declarative way – a framework that takes the constraints and applies them for you, or they can be implemented in an imperative manner – a set of programs that run in a defined order to check

the constraints. To explain the advantages of the declarative way of working over the imperative way of working, let's start with a programming example.

In a recent blog post at Cocoa With Love — a blog about writing apps for macOS and iOS — Matt Gallagher went into why declarative views are preferable above imperative views. In it he stresses that in an imperative system, the context of the life cycle of your object is important when deciding which rule to run. He contrasts this to a declarative system, where all rules are declared up front, and where all rules are **always active**.

He states:

> *In a declarative system, all of this goes away because your rules are always true. The lifecycle still occurs but you don't need to make decisions based upon it. Your rules will be applied automatically, at the best possible time.*
>
> *Since you don't need to understand how the declarative system works, you're not as dependent upon its minor details. Fewer dependencies make code easier to abstract and consequently, you can make improvements at every level.*

The same distinction between imperative and declarative rules holds for data models. The rules specified in a logical data model are always true; their context is inherent in the data model structure.

Or, as Matt Gallagher states:

> [In a declarative system] *rules and relationships cannot be changed during the lifetime of the system (they are invariant), so any dynamic behavior in the system must be part of the description from the beginning.*

Let me give an example on how the above statements hold true for data models. Take an attribute that is **declared** as primary key. It is always true that this attribute identifies a specific tuple within its entity type. There is no point in the life cycle of the tuple where that attribute cannot be the primary key. There is no point in the life cycle of the entity type where that attribute is not the primary key. This declaration of the attribute as primary key "travels down" from the logical model, via the physical model, to the technical implementation. This mechanism is called the "conservation of concerns".

Of course, you can make an **imperative** rule out of the fact that an attribute is a primary key. When doing that, you have to write a program that is triggered to run after the tuple has been inserted or updated. This program has to check that the values of the field in the data base that represents this attribute is unique, that this value is not used more than once. Also, when doing bulk inserts or bulk updates, that program has to run, and it has to run after the life cycle of the whole entity has stabilized. When forgetting one specific

life cycle state, or simply when the program does not run due to an error, the data integrity of your data base will suffer.

And here, the second part of Matt Gallagher statements about declarative programming also holds for data integrity and data validations. You don't need to understand how the declarative system works and you are not as dependent on its minor details. Fewer dependencies make the model easier to read and the abstraction level of the logical data model, physical data model and technical implementation ensure that improvements can be done at the appropriate level.