How to define Entities

The last 15 years, methodologies promoted **Process** modelling reducing importance of **Entity** definitions. To day it is more and more recognized that definition of Entities is the first step to define an Enterprise System, even if Process Approach brings high value.

1 Abstract

Why define Entities?
Entities must be defined for **clear requirements**: it is obvious that to define Business Processes like "Create a Person" or "Subscribe a Contract" the first step must be to define the Entities "Person" or "Contract", and the most common words are the most difficult to define because they generally represent different Entities depending on who use the word.
Entities must be defined for **Mutualisation**: share Processes, share Products, share Data, share Exchanges or share Software Services, means sharing a common language.
But definition of Entities is is also a necessity to build a **good software**.
Each Enterprise System includes thousands of **Attributes** and **Business Rules**.
A clean structure of Entities is the best way to classify these Attributes and Business Rules.

Which common Attributes for Entities
Any Entity must have

- An **Identifier** which identify each instance of the same Entity: be careful to choose invariant identifier
- A **Version** to make a distinction between the different image of the same instance across time
- **Who** and **When** for last image
- **Status** which is linked to life cycle

Approach to define Entities
Define Enterprise scope: definition at Group level, company level or lower level?
Define Entity Domains: Actors, Products, Contracts, Organization,
Look for existing Glossaries: internal or external glossaries (from inter professional initiative or package provider)
Define **links** between Entities: relation ("has") and inheritance ("is"). Progressively build the Entity Model: Entities are defined altogether.
Define **Descriptors**: Entities with simple life cycle (like Person, Address, Organization, …), also called referentials
Define **Operations**: Entities which represent Inputs (they must be prepared then executed).
Define **Multi Execution Processes**: Entities with complex life cycle which represent the more complex activity of the Enterprise

Thumb rules to define Entities
Do not detail the data model before having built a complete Entity Model.
Start defining Static data (referential data like Actors, Organization, Products), then go to more dynamic data.
Define owner relations.
Prefer relations between instances of the same Entities rather than building new Entities.
Prefer a multi-status Entity rather than N Entities
Provide identifier for each Entity
Start with concrete Entities before abstract Entities
For international definitions, reuse same Entities
Start with Parent Entity before Child Entity
Manage Synonyms (customer/client, contract/policy, ...)
Manage Homonyms
Do not mix inheritance (« is ») and relation (« has »)
Define relations with «creation process»
Do not mix Entities and Business rules
Split containers and contents
Check that existing data find a place in the new Entity Model

How to transform Entities into the right Classes

Once defined, Entities can be implemented with or without Object Oriented tools which allow to benefit from mechanisms like inheritance.
The **same Entity** can be implemented with **several Classes**. Example: the Entity “Order” is implemented with the Class “Order header” and the Class “Order Line”. The Class “Order Line” is **owned** by the Class “Order Header”. It means that Entities and Classes are not he same: how to go from Entities to Classes?
Transform **relations** carrying information into attributes or classes
Be careful not to be **too generic** : when an Entity is difficult to analyze, one easy solution is to define a very general Entity and give it many different meanings, like a list of “Person+ the Role they play”. Try to be pore precise at design time to avoid useless complexity for Developers who use the Entity.
If a Class becomes **too large**, split into several Classes.
Build and **certify** the **Class Model** which presents Entities with relations and inheritance links.
Define **precise** Relations.
Implement a **Role** as a Class or an Attribute.
Prefer several Classes rather then different **Perspectives** of the same Entity

Approach to help Business Analysts and IT Developers to reuse the Entity Model
Provide an **Entity Model** (definitions, relations, inheritances, identifiers, Life Cycle and Versioning) **before** providing a detailed Data Model.
Do not communicate the **Semantic** Model to Business Analysts and IT Developers.
Propose **Services**: training, coaching, and certification.
Propose a **tool** to help Business Analysts and Developers to search for Entity definitions, and administrate them.

2 Objectives

2.1 Entities must be defined for clear requirements

Most of our sentences are structured the same way: **Subject + Verb + Object**.
Example: the Salesman Sells an Insurance Contract.

When Business Analysts describe the activity which they want to computerize, they use the same common language: the only difference is that they prefer

- the word "**Actor**" instead of subject (for "Salesman")
- the word "**Business Entity**" instead of Object (for "Contract")
- the word "**Process**" or "**Activity**" instead of Verb + Business Entity (for "Sells a Contract")

So the question is: how to define Actors, Processes and Entities?
Our recommendation is to define them the **reverse order**:
1-define the Entities
2-define the Processes
3-define the Actors and who does what

Why to define the Entities before Processes?

It is obvious that to define Business Processes like "Create a Person" or "Subscribe a Contract" the first step must be to define the Entities "Person" or "Contract". But the most common words are the most difficult to define because they generally represent different Entities depending on who use the word

Why to define Actors after Processes?

We suggest defining requirements in 2 phases:

- first define "**Business Processes**" which represent the "**what**" must be done,
- then define **who** does what: for the same Business Process, many scenarios with different combinations of Actors can be described. Each of them is called an "**Organization Process**".

For example the Business Process "Subscribe a Contract" may have several Organization Processes:

- the Salesman Subscribes the Contract (on his desktop, or laptop)
- the Customer Subscribes the Contract (on Internet)

2.2 Entities must be defined for mutualisation

More and more Enterprises are willing to Mutualize at different levels: share Processes, share Products, share Data, share Exchanges between the different IT Blocks, or share Software Services.
It will be impossible to share if there is no common definition of the Entities involved in this sharing.

2.3 Entities must be defined for a good software structure

An Entity represents an Object of the real world.
An Entity has an **identifier** which allows to recognize the different instances of the same Entity.
Good example: a Person, an Account, a Product, a Contract, an Address, …
Bad example: a Report, a Window, …
For each Entity a clear **definition** must be written, as explained.
But definition of Entities is a necessity not only to use a **clear language**, it is also a necessity to build a **good software**.
Each Enterprise System includes thousands of **Attributes** and **Business Rules**.
A clean structure of Entities is the best way to classify these Attributes and Business Rules.
*When the definition is clear, we generally create one **Class** for each kind of Entity: a Class is a piece of software which mainly contains **Data** (called **Attributes**) and **rules** (called "**Methods**") of the Entity.*

3 Which common Attributes for Entities?

Each Entity owns its attributes, but some Attributes are always present for each Entity:

- Identifier
- Version
- Who and Where
- Status

3.1 Identifiers

The main characteristics of identifiers are:

- Do not mix different instances of the same class: the simplest solution is to increment a number by one at each instance creation.
  - But what if a salesman creates instances of Contract on his laptop if it is not connected: number slices or use laptop id as identifier header?
  - But what if an instance changes: how to address the right version number
- **Internal Id = name space + order**
  - Name space = who generates the order: generally it is an Organization Unit or a System
    - Ex: the mainframe of a company
    - Ex: the WS (including laptop)
    - Ex: the server of a country
  - Order: +1 for each new id (do not reuse an id even if instance is deleted)
- Most of Entity instances are updated several times during their life cycle . To identify an image of an Instance at a given time, the identifier is not sufficient: we must add a **version** number. For each update, the version number is incremented and the begin date is kept. Each image is then accessible by version number or by date. Rules must be defined for each Entity to describe which versions are kept alive in the Enterprise System.
- All relations must be done through this identifier. Version is or is not present depending on the context.
  - Sometimes, we prefer to access the last version. Ex: last version of the Address
  - Sometimes we prefer to access a given version. Ex: version of a contract (by number or by date)
- **Access key:** when the identifier is not known, the Object must be accessible by other ways which generally are some values of attributes, called “access keys”. Ex: access to a Person by name, first name and birth date
- **Specific identifier**: for some Entities, already exists an identifier which must be reused. We advice to keep the internal id to benefit from all common services, and to add the specific identifier as another key
- **Internal ID or Partner ID**?
  - For broker AON, 3 ID are kept: technical ID, Shared ID and External ID (for AXA, AGF, …)
- **Name**: most of the Entities have a Name (name of a Person, a Legal entity, a Product…), which is a string that the users remember better than identifiers. Instances can be found by name; if homonyms, selection must be done with other complement data of the instance.

3.2 Versions

An Entity changes during its lifetime.
This is why we advice to add the version to the identifier.
Version is incremented by one at each update.
Rules must be established to define which versions are kept in the system or delete or archived, like:

- Keep all versions
- Keep all versions for a given period (all the versions from last year)
- Keep all versions on first of month
- Ask the user to determine if he wants or not to keep the last version for each update

- …. and many other strategies

Versioning solves the **history** problem for any instance.

3.3 Tracking information

For each version, keep track of:

- **Who** is responsible for last update
- **When** was it done
- ("where" is not required)

3.4 Status

One of the most important Attribute of an Entity is its **Status** because it explains where the Entity is in its **Life cycle**.

Each Entity has a life cycle. To summarize what was defined above:

We define **simple** life cycle for following Entities

- Descriptor (like Person, Address, Organization, …) : create, modify, delete, archive
- Stock (like Account, inventory): create, modify, delete, archive + debit + credit + explain balance
- Operation (like "modify Address", "create a customer", "Transfer money": create, suspend, modify, authorize, execute, delete, archive

Some specific **Complex** life cycle exist for more complex Entities like: Loan in a Bank, Claim in an Insurance Company, or software update for all (we called them Multi Execution entities because they require several Executions during its life cycle).

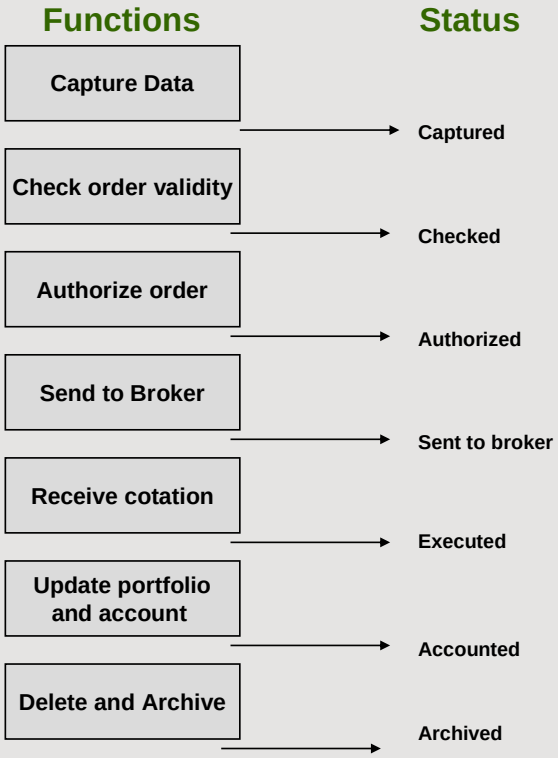
To explain ow Status and Life Cycle are linked, let's take the example of the "Stock Order" life cycle. The successive Functions of the Process are:

- capture data,
- check order validity,
- authorize operation,
- send to Broker,
- update portfolio and accounts,
- delete and archive

For each Function of the Life Cycle, a **status** is updated.

- captured,
- checked
- authorized
- sent to broker
- executed
- accounted
- archived

Status and life cycle: stock order Process example



Page 25

Values for Status

To simplify language between Business Analysts and Developers, we must try to standardize the **common** values of status like

- Created but not in force
- Valid
- Suspended
- Deleted
- Executed
- Archived

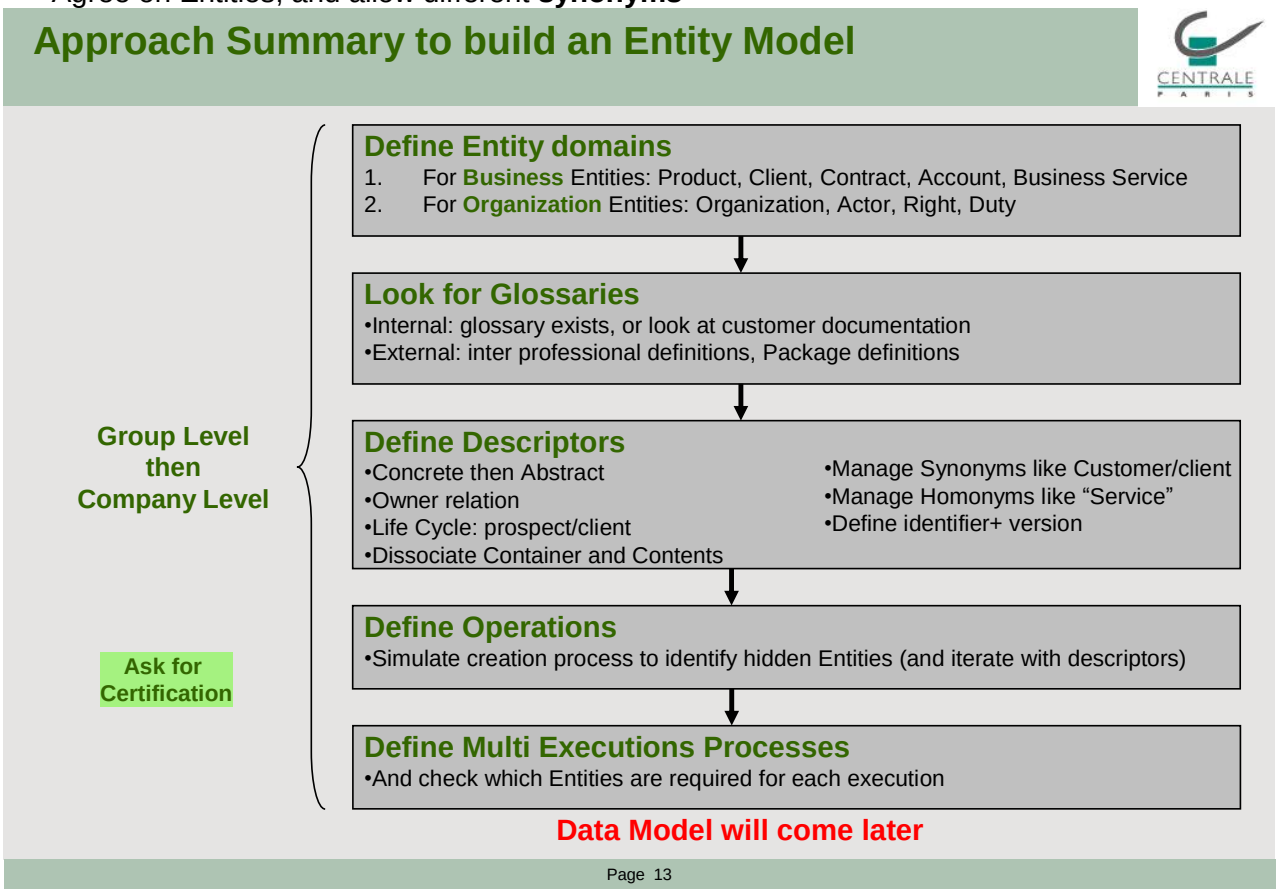
Then add **specific** status values for each Entity.

Ex: for the Entity “Prospect” add values : “identified”, “identified and contact done”, “interested”, “cancelled”

4 Approach to define Entities

Every designer generally thinks that he is able to design a good Business Entity Model, while it certainly is one of the most difficult tasks which requires a lot of experience.
CEISAR suggest that, for the first times, the designers should ask for help: find real experts, copy existing models and improve them, ask model experts to **certify** the quality of the Entity model and reach a high level in modeling.
Defining words is a very difficult task. People who write dictionaries know that.

- A definition must be **unambiguous**: based on the talent of the writer, and on the quality of the other definitions
- A definition must be associated to an **example**
- A dictionary is better managed by a **single person**, even if many contribute.
- A definition must be as **short** as possible as long as it is understood
- As definitions reuse words already defined, it is sometimes necessary to update the definition of an already defined word: it is an **iterative** process
- **Reuse** already existing definitions, do not reinvent new terms
- Agree on Entities, and allow different **synonyms**



To help you to start, find below some practical rules, based on experience, starting with rules on "Entities", then rules on "Business Classes".

4.1 Group level definition or local definitions?

If the Group wants to share

- Data (like customer file or Business Intelligence data)
- Ideas (and even solutions) on good Products or good Processes
- Software Blocks
- Software Services

then, it is better to have defined a common glossary for Entities. **Group** level will be less detailed than Company level which reuses Group definitions and add complements like for Russian dolls.

But, be careful not to grow too much the Group Model which will be reused by each company of the Group: if it is too complex, companies will not reuse it.
Use an **iterative** approach to refine the Model in successive versions. It allows everyone to improve the common Model.

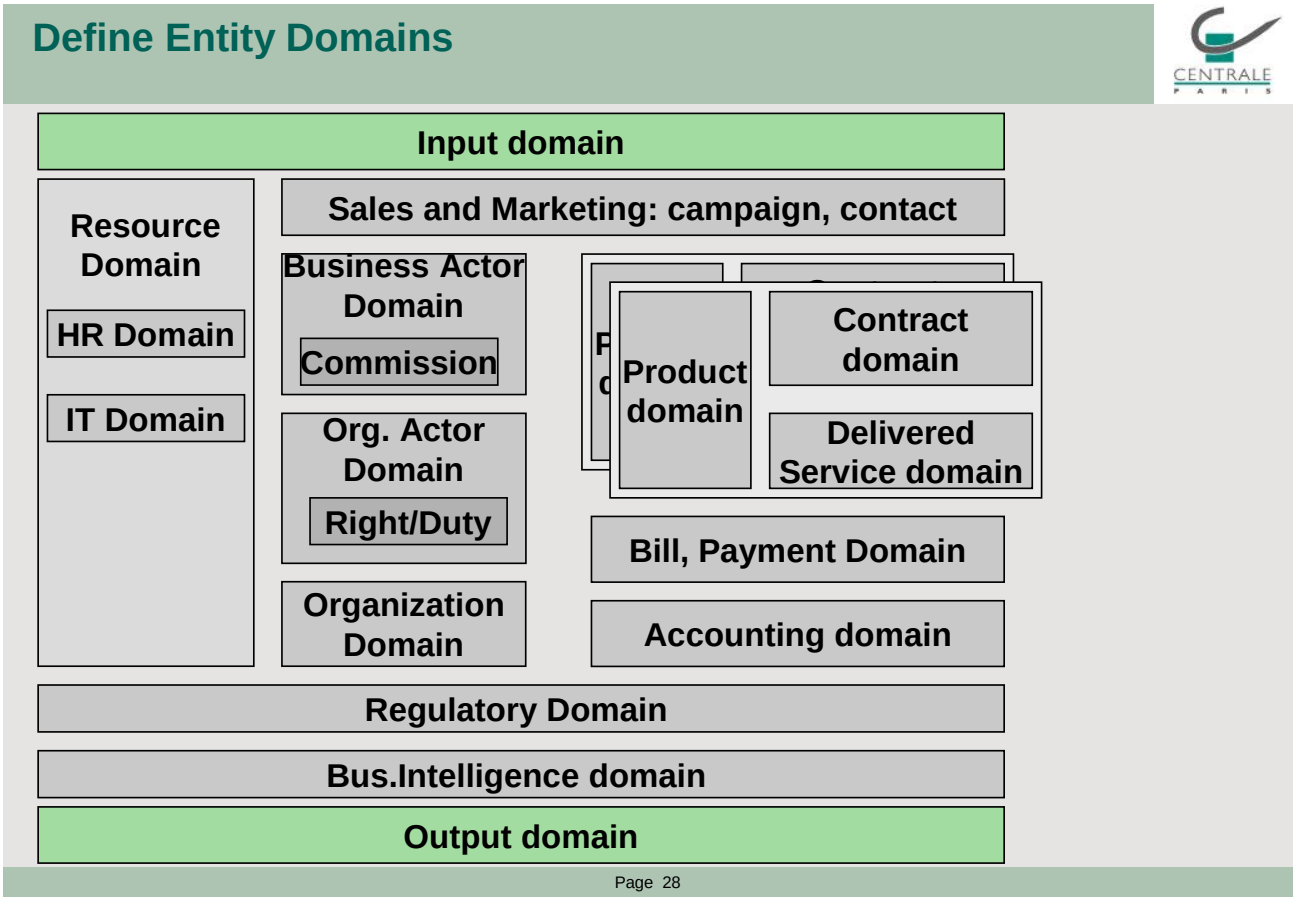
- Do the work at different levels:
- Maintaining this Entity Model is the role of the **Business Architecture team**.

4.2 Find main Entity Domains

Simply describe the activity of the Enterprise and give a definition to all important nouns.

- Who is the Customer
- What is the Product (goods and services) offered to the customer
- With what Resources: people, machines, organization, location
- What is necessary to Sell: contract, distributor
- What is necessary to Deliver the product or service: Beneficiary, Provider, Delivered Service

Start with Business Entities, then define Organization Entities.



4.3 Look for existing Glossaries

4.3.1 Look for an internal Business Glossary

The first place to look for glossary is the company itself. But it is very disappointing to notice that very few companies have formalized what is a Customer, a Product, a Contract or a Service.
Another way to understand the language used in the company is to:

- read documents defining products or processes and try to extract the definitions
- analyse **input and output documents** to look for complementary entities : customer documentation (marketing, contract, …), partner documentation
- talk to business people and IT people in charge of related domains

4.3.2 Look for an external Business Glossary

Other sources may come from:

- **inter-professional organizations**: to exchange with their members, they are using a language which is understood by all, like EDI definitions,
  - Example for Pharmaceutical Industry different organizations have defined a language: Efpia (Business Europe), Leem (Business France), Cedhys (IT France), EdiPharm (EDI France)
  - SCORE for supply chain
  - OAG for manufacturing
- **Application package** providers: their documentation define the Entities
- **Technology providers** like IBM
- **APICS for industry, supply chain**

Many Standards exist today, not always very consistent between them.
To help you, CEISAR has defined a first list of Entities, as an example.

4.4 Progressively build the Entity Model: links between Entities

The textual definition of an Entity reuses other Entities.
Example: "The **Contract** is the formalization of an agreement when a **Legal Entity** sells a **Product** to a **Subscriber**", means that a Contract has links with other Entities: Legal entity, Product and Subscriber.
Formalization of these links will help to build definitions. Two kinds of links are useful: inheritance and relation.

4.4.1 Inheritance: "is"

Class B inherits from class A when it **is** a specialization of Class A.
Class A is the **Parent** Class. Class B is the **Child** class.
Example:

- a "Life insurance policy" **inherits** from an "Insurance Policy",
- a "Term Life policy" **inherits** from a "Life insurance policy", which means that inheritance is a hierarchy.

Inheritance helps reuse the common part of a policy: a child class benefits from the data and rules of its parent, grandparent, great-grandparent, classes.
Use inheritance when you can use "**is**" as a verb between classes (such as a Term Life Policy is a specialization of a Life Insurance Policy).
One of the most powerful features of inheritance is that when class A evolves, modifications are automatically applied to inherited class B, which is not the case with simple cut and paste.
Another powerful feature is polymorphism in which an entity can manipulate an instance by invoking methods of its parent class without any knowledge of the specialized class.

4.4.2 Relation: "has"

A Person **has** an Address. A Customer **has** contracts. An Account **has** Account Lines.
Each of them is called "Relation".

4.5 Understand and reuse the business patterns: Descriptor, Operation and Stock

Each Entity is unique. But they can be grouped by **behaviour** or **life cycle**.
We generally define 3 basic behaviours: Descriptor, Operation, Stock, the others being called "multi execution Process".
It is sufficient to define that a Person is a Descriptor to implicitly mean that we need Processes like:

- "Create a Person"
- "Delete a Person"
- "Modify a Person"
- "Search a Person by identifier"

This is a good way to simplify requirements.

4.5.1 Descriptor

A Descriptor is a Entity whose Life Cycle is simple: create, modify (a new version), interrogate, and delete.
Some call it “Static Entity”.
Ex: “Person”, “Address”, “Organization Unit”, …
Many Entities behave this way like: Person, Legal Entity, Organisation Unit, Address …
When you define the Entity “Person” as a Descriptor, there is no need to redefine its life cycle: just add to standard Descriptor Data (like id, version, by who and when it was updated) the specific Data.
Descriptors may be reused by different Processes.

4.5.2 Operation

An Operation is an Entity whose Life cycle is different.
Some call it “**Dynamic Entity**”, or “**Input**”, or “**Event**”.
Ex: “Change Address”, “Order”, “Transfer Money”
An Operation is **prepared** in one or several tasks by one or successive application users.
When an Operation is executable (validity of data and authorized), it can be **executed**, which means that the irreversible actions like updates to other objects can be performed. An operation can be executed just once. After being executed, the operation can be kept persistent or archived, but is never modified.
The main difference with Descriptor is “**Execute**” which represents the set of irreversible actions of the operation.
Example: money transfer.
You create a money transfer, you can suspend it because you lack the account number, or you need someone else to authorize it. But at the end, either you abandon the money transfer or you **Execute** it which means that you transfer the money, update the account and print an order copy. After Executing the Operation you cannot change it anymore: you can only keep it persistent in the system to let people interrogate the operation until you decide to delete and archive it.

4.5.3 Stock

A Stock “is” *(which means “inheritance”)* a specialization of a Descriptor which has a **balance** and has relations with Operations which updated the Balance.
Example: the Entity “Account” is a Stock. Its balance is credited or debited each time there is an accounting operation. Idem with Inventory.

4.5.4 Multi Execution Process

Some Entities have a more complex life cycle.
For example a “Loan Contract” requires to be **executed several times**, which means that there exist successive irreversible actions like:

- Record customer request and get all required papers
- Get approval from Bank management
- Get approval from customer
- Put the loan contract in place at the right date

Each time the Loan Entity is executed, it changes its **Status.**
These Multi Execution Processes are better managed through workflow engines

5 Thumb Rules

5.1 Entities or data model?

Classical methodologies insist on describing independently data and rules.
“data” generally means Entity definition, tables, their attributes, and relations between tables.
We suggest to improve this classical way by 2 principles:

- Use inheritance (see above)
- Define **Entities before detailing data inside**.
- Do not try to centralize detailed data definitions at Group level: if you propose a super set of the different data which could be useful, it means that you will produce huge quantity of information that scares people. They do prefer to be helped on getting well structured Empty boxes, and fill them by themselves with the data they require.

5.2 Start defining static Entities then go to more dynamic Entities

This rule comes back to: first define Descriptors and Stocks, then Operations (mono execution process) and Multi Execution Processes.
In some Organizations, Entities which are not often updated but often read are called “**Referential** Data”. They generally include: Actor, Organization and Product.
As these Entities are reused by many others, they generally are defined first, which is easy for Actor and Organization, but not so easy for **Products**: we advise to define Product together with Contract and Services because they must keep consistent.

5.3 Define owner relations: tightly coupled Entities

An instance of a **dependent** Entity cannot exist if it is not linked to an instance of the **owner** Entity (in UML we use the “aggregation” word).
Example:

Owner Entity	**Dependent** Entity
Legal Entity	Address
Person	Medical Status

The rule should be

- Find owner Entity
- Find relations between owner Entities
- Add dependent Entities

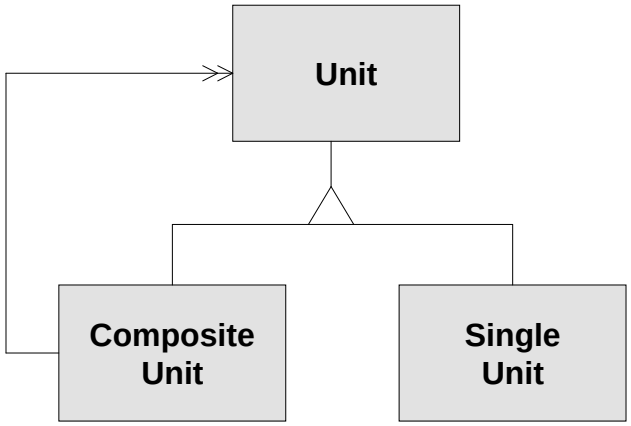
Start with Entities which are not related to other Entities
Ex: do not define “Address” which cannot exist by itself; start defining Person, Legal entity, then define Address which will be related to Person or Legal Entity.

5.4 Prefer relations between instances of the same Entities rather than building new Entities

If your **Organization** is built with different levels like “Direction”, “Department”, “Division”, “Region”, “Branch”, do not create as many Entities as there are levels: just create one Entity called “Organization Unit” which has an attribute “Level of an organization unit”, and create relations between these Organization Units.
Same thing if you want to describe **premises**. One Entity “Location”, and one attribute “level of the Location” which can get values like “country”, “region”, “campus”, “building”, “story”, “room”.

Other example: a **Material** composed of Materials.

Organization, Premises, Materials, “Urbanism” Blocks are **hierarchical** structures; they are generally represented with the following model:



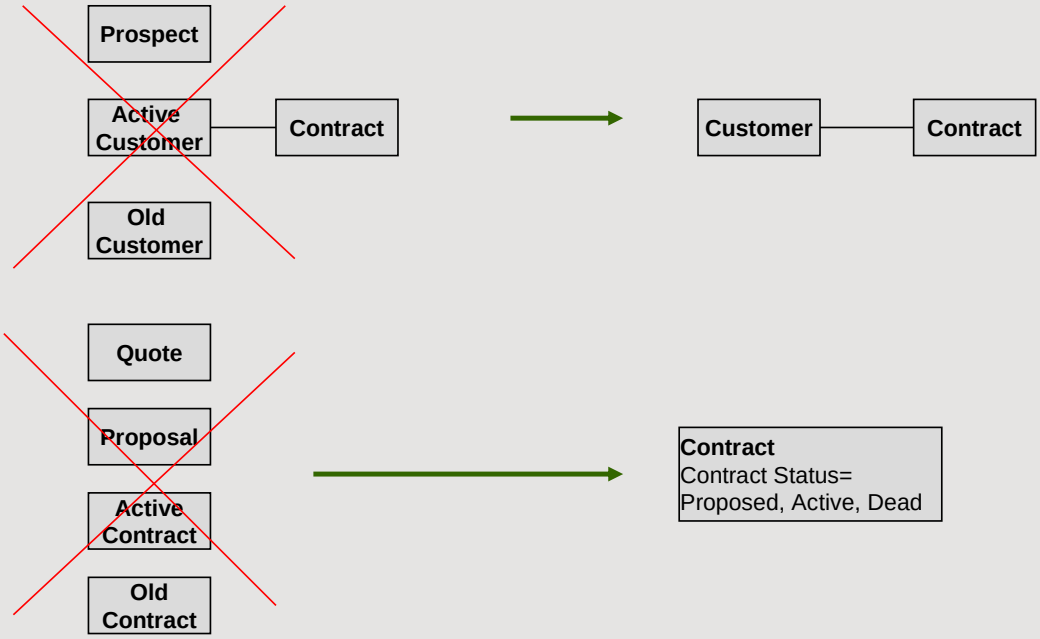
Some other internal relations are not hierarchical like:

- a Person related to other persons (family relations)
- a Legal entity related to other Legal entities (ownership relations)

5.5 Prefer a multi-status Entity rather than N Entities

A **Proposal** is the same Entity than the **Contract**. It is better to define the Entity contract and a status which indicates “proposed”, “subscribed”, “suspended”, “cancelled” …
Same thing with “**Prospect**” and “**Customer**”. The Entity is Customer, the status is “has no active contract”, “has active contract”, “had active contracts”. This solution is better because it solves the ambiguity when asking questions like “*How many Customers”,* must we include old Customers, Prospects?

Avoid multiple Entities



Page 14

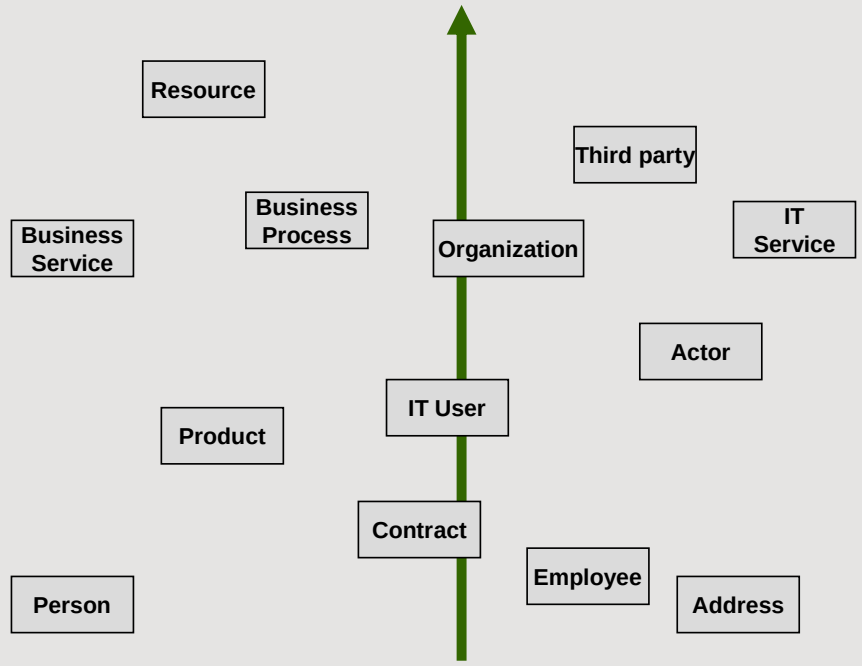
5.6 Provide identifier for each Entity

- Define an identifier which does not mean anything, so that it can be kept during all the life cycle. Significant identifiers make problems like:
  - Account number with branch id: what if a customer changes its Branch?
  - Car with department number or State id: what if a car is sold and changes department or state?
- Include versioning:
  - with dates
  - with tracking information
  - with cleaning rules (which versions must be kept)
- Define which **IT Resource** generates the id: generally called “name space” (Ex: Server id or Laptop id). Example: non connected laptop used by sales people allow to generate definitive Contract Identifier because identifier includes the Laptop id.

5.7 Start with concrete Entities before abstract Entities

Start with **concrete** Entities which are easy to define like a Person, a Contract (there is a signed document), an Address, an Employee…
Then define **abstract** Entities like Third Party, Resource, Actor …

Concrete before Abstract



Page 15

5.8 For international definitions, reuse same Entities

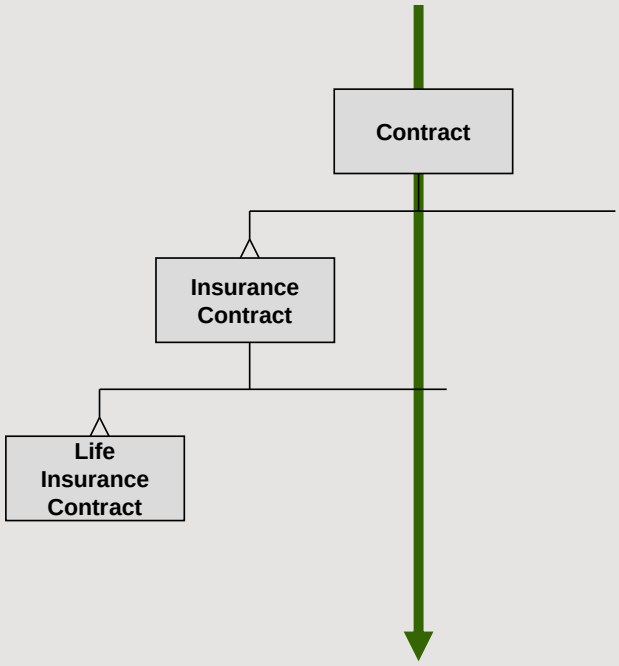
Each Enterprise in each country thinks it is specific and requires specific Entity definitions.
But cultures, products and processes get closer and closer.
We suggest to reuse same Entities and to implement specificities through **different** data and rules in **same** Entities.
Define Entities for one country, validate first version of the model with another country and progressively improve the **common model**.

5.9 Start with Parent Entity before Child Entity

A Child Entity is defined from the Parent Entity. Ex:

- First define a general **Contract** as an agreement between a company and a customer. Attribute examples: id, date of subscription and subscribed Product
- Then define its child: an **Insurance contract** is a **Contract** where the Company is an Insurer. Attribute examples: what is in Contract + the insurance agent who sold the contract
- Then define a **Life insurance Contract** as an **Insurance Contract** for a life product. Attribute examples: what is in the Insurance Contract + the coverage amount.

Parent before Child

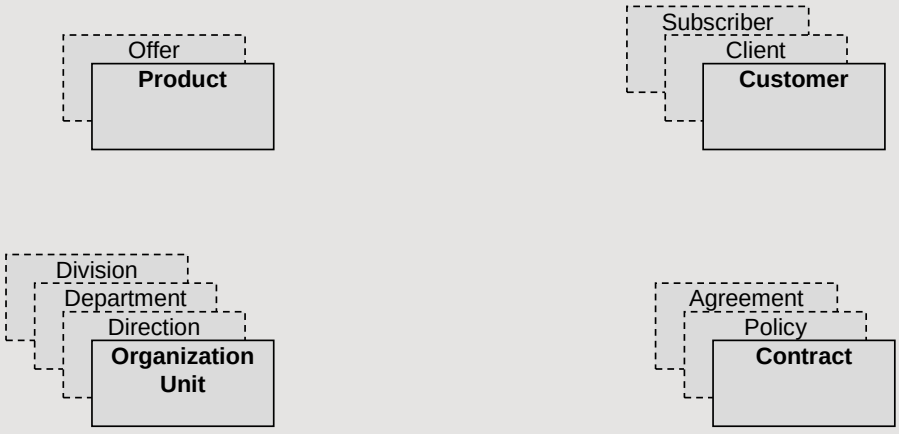


Page 16

5.10 Manage Synonyms (customer/client, contract/policy, ...)

It is much more **difficult** to have **few** Entities than **many** Entities. Finding that an Entity looks like another one requires a lot of attention. The talent of the designers is high when they define a low number of Entities.
When Business Analysts describe their Processes, they always like to create new Entities because they consider that their Business is always very specific.
To limit the number of Entities, the first simple rule is to **allow Synonyms**.
For example create the Entity “Customer” and allow to call it “Client”.
Or create the Entity “Contract” and allow to call it “Policy”.
The ideal system would allow each company to define which term they want to use for each Entity, and automatically generate a documentation which only use the preferred term.

Manage Synonyms

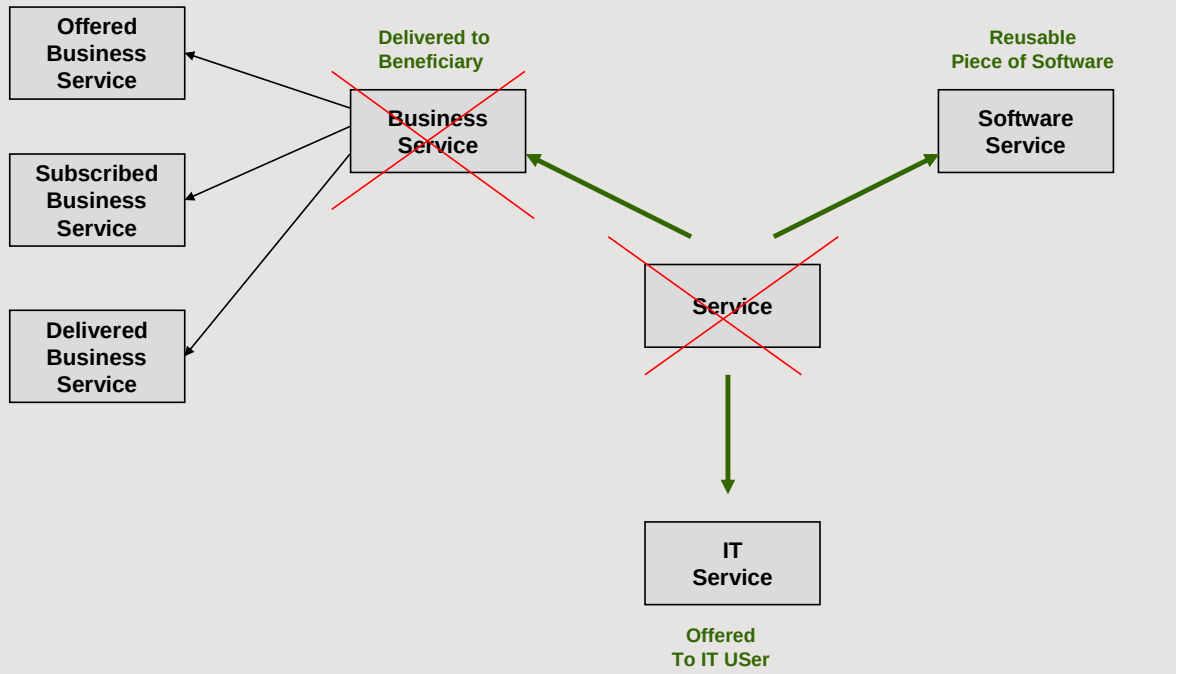


Page 17

5.11 Manage Homonyms

This is one of the most difficult problems: people often attach different meanings to the same word. The only help we could provide would be to offer a **documentation tool** which, while typing, dynamically recognizes that a word has already been defined and proposes to the writer to dynamically visualize the definition to check that he really agrees on it.
If the meaning is different then either he changes the term, or he keeps it, which means that the documentation system must then remember (through indices like in a dictionary), that there exist several homonyms for the same term, and next time the same term is recognized, several definitions will be proposed.

Manage Homonyms

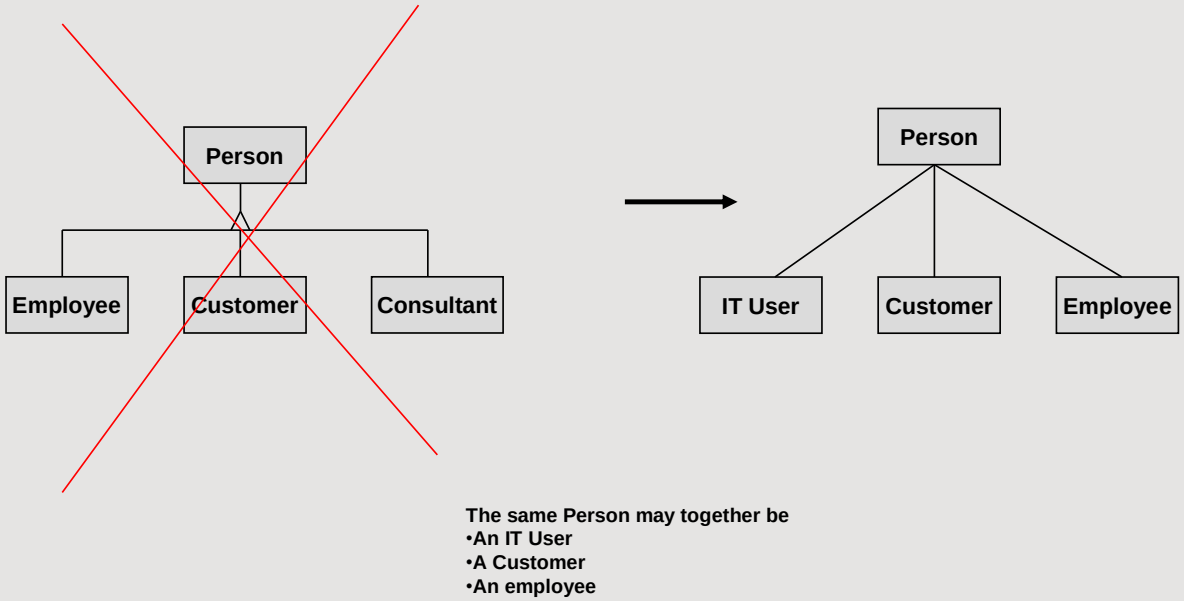


5.12 Do not mix inheritance (« is ») and relation (« has »)

This is a current mistake. It comes from the **limit** of the “is” definition.
Let’s give an example: when you define the Entity “**Employee**” because you want to computerize payroll, you can say “an Employee **is** a Person”: so “Employee” inherits from “Person”.
Then you want to implement an authorization system and you want your employee to get “Rights”: you create User id and password in the Employee Entity.
Then you decide that your **customers** may also access to your information system, and so will get id and password. But as a Customer is not an Employee you create a Business class which inherits from Person and you include id and password which do not follow the same rules. As your Employees are also customers, they must get 2 sets of id-password.
Then you want that your **consultants** have also access to your information system. But as they are not employee, you must another time include id and password with different rules…

To avoid this complexity, the good rule is to **use relations and not inheritance**: a Person (or a Legal Entity) may play different **roles**.
The Employee is a Person who **has** Employee Data and rules.
The IT User is a Person who **has** id/password data
The Customer is a Person who **has** Customer data
And the same Person may be together Employee, User and Customer.

Inheritance or Relation?



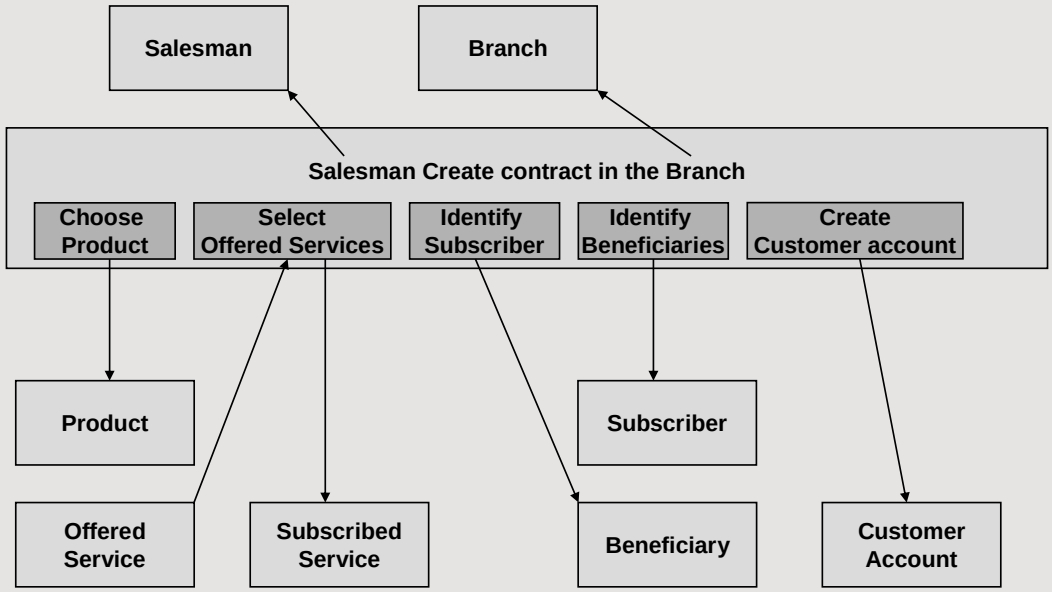
The same Person may together be
- **An IT User**
- **A Customer**
- **An employee**

Page 13

5.13 Define relations with «creation process»

To help find Classes, another way is to define the creation process of each main Entity, and look for related classes.

Find Entities through creation Process



Page 19

5.14 Do not mix Entities and Business rules

For example, a “condition” is not an Entity, but a Business Rule inside an Entity.

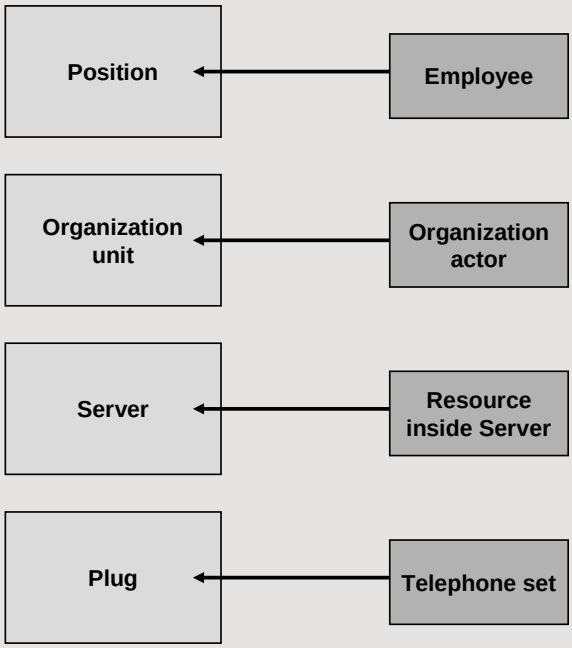
5.15 Split containers and contents

There is often a confusion between a container and a content. For example:

- an **Employee** (Mrs Smith) and a **Position** (the Position of Assistant occupied by Mrs Smith)
- a telephone plug and a telephone set
- a hardware and a resource inside the harware
- an Organization Unit and the people inside it

To explain why this distinction is useful, let’s take the example of “**Employee-Position**”.
To define who will take care of each Customer, we can use the Employee: it means that in each Contract, we will create a relation towards “Employee”. If the Employee 1 moves and is replaced by Employee 2, then many Customer Objects must be modified to replace “Employee 1” by “Employee 2”, a lot of work.
If we use the Position “Sales People Position”, then there is nothing to modify when the Employee moves, except the relation from Position to “Employee 1” which then relates to “Employee 2”: much lighter!

Container and content

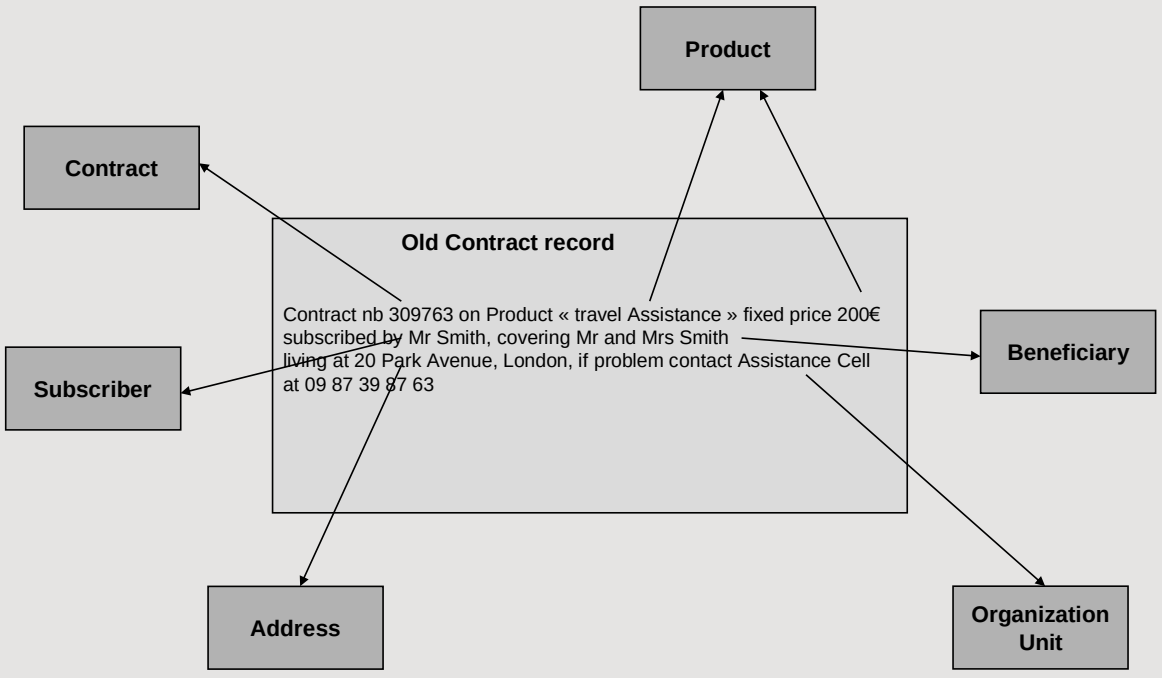


Page 20

5.16 Check that existing data find a place in the new Entity Model

This check is easy. If some data cannot be localized into the new Entities, then either this data is useless, or an Entity was forgotten.

Check that each old data find its place



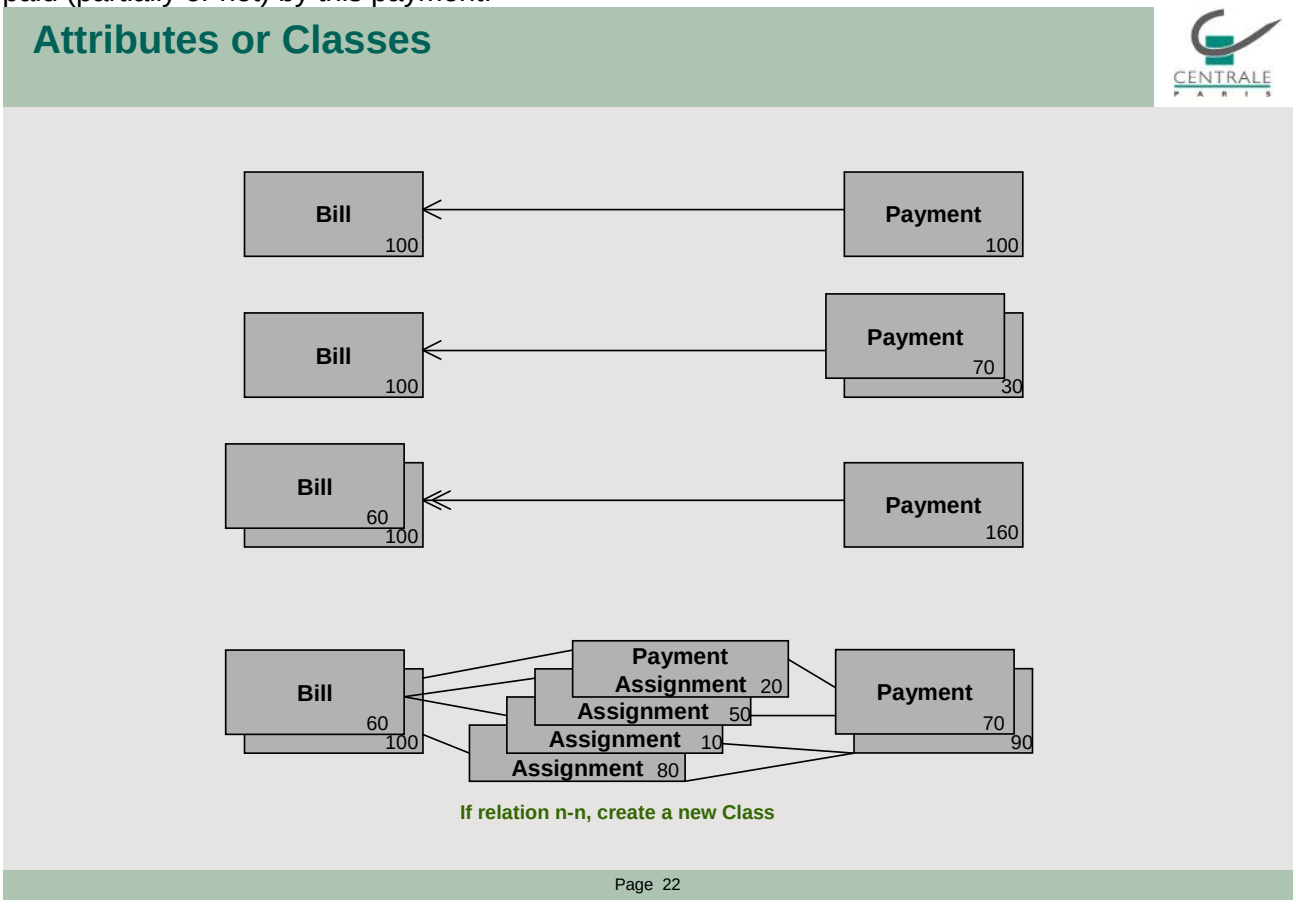
Page 21

6 How to transform Entities into the right Classes

Once defined, Entities can be implemented with or without Object Oriented tools which allow to benefit from mechanisms like inheritance.
The **same Entity** can be implemented with **several Classes**. Example: the Entity “Order” is implemented with the Class “Order header” and the Class “Order Line”. The Class “Order Line” is **owned** by the Class “Order Header”. It means that Entities and Classes are not he same: how to go from Entities to Classes?

6.1 Transform relations carrying information into attributes or classes

If one Bill is paid by one Payment, there is a single relation from Payment to Bill.
If several bills can be paid by a single payment; then there is a multiple relation from a Payment to the different Bills.
If the same Bill can be paid by several Payments, then there is a multiple relation from the Bill to the different payments.
If several Bills are paid by several payments, then a new class must be created which will be called “payment allocation”. A Payment will be related to as many “payment assignment” that there are bills paid (partially or not) by this payment.



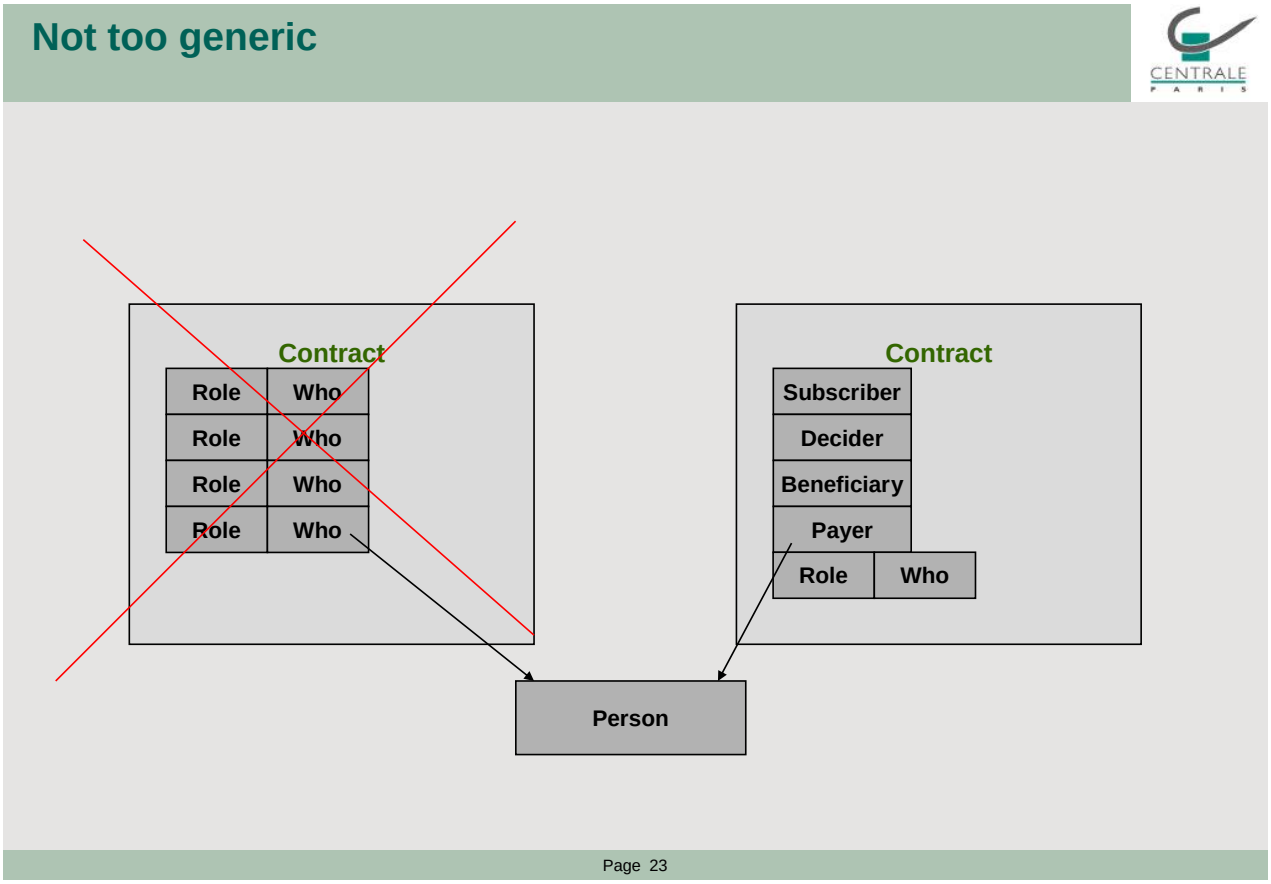
6.2 Be careful not to be too generic

When an Entity is difficult to analyze, one easy solution is to define a very general Entity and give it many different meanings.

Example 1: for the Entity Contract we need several Actors like a Subscriber, a Payer, one or several Beneficiaries.

The Business Analyst may say “well, may be I will forget other Actors related to Contract; so let’s implement it with a very general concept: a list of (Person + the Role they play)”.
By doing so, a big advantage is that you can add other Actors if you have forgotten some.
But a big disadvantage is that each time you require a specific Actor, like Subscriber, you must look successively to all Actors and select the ones which have the Role “Subscriber”.
It also means that consistency will be more difficult to implement. For example, if there is one and only one Subscriber by Contract:

- it is easy to implement if you define one attribute “Subscriber” which is a relation from Contract to Person which is a single relation (no multiple values) which cannot be empty
- it is more difficult if you use a list of “Person + Role”: you must then check through the whole list that there is one and only one Subscriber Role, and the application developer will have to write code for it.



Example 2: for the Product structure, some company prefer to define a single Entity for Product, or Contract, or Delivered Service, with as many relations as possible. This allows any product structure, but makes the implementation much more complex, because the developer cannot take advantage of the Business structure.

Example 3: a Person and a Legal Entity are very different Entities, but some companies like to merge them in one general concept of “Person” because Contract can be subscribed by a Person or a Company.

6.3 If a Class becomes too large, split into several Classes

When your Class has too many data or rules, it becomes difficult to manage: then split it into several Classes.
“Too many” means more than 20 Variables, or 20 Rules, or more than one page of code.

6.4 Class model

The Model which represents the classes with the links (inheritance and relation) is called "**Class Model**". He represents the backbone of the application software.
Once this structure exist, it is easy to progressively fill each Class with the right Attributes and Methods.
If the quality of the model is high, it really is **easy to add** new business functions, new data, new GUI and it is easy to obtain good performance and reliability.
So the most important objective is to build the application on a high quality model. If some business functions are lacking in the first version of the application, it will be easy to add them in following versions.
Ask an expert to **certify** the Class Model: it seems easy to build it and everyone thinks he is able to do a good job while experience shows reverse evidence.

6.5 Define precise Relations

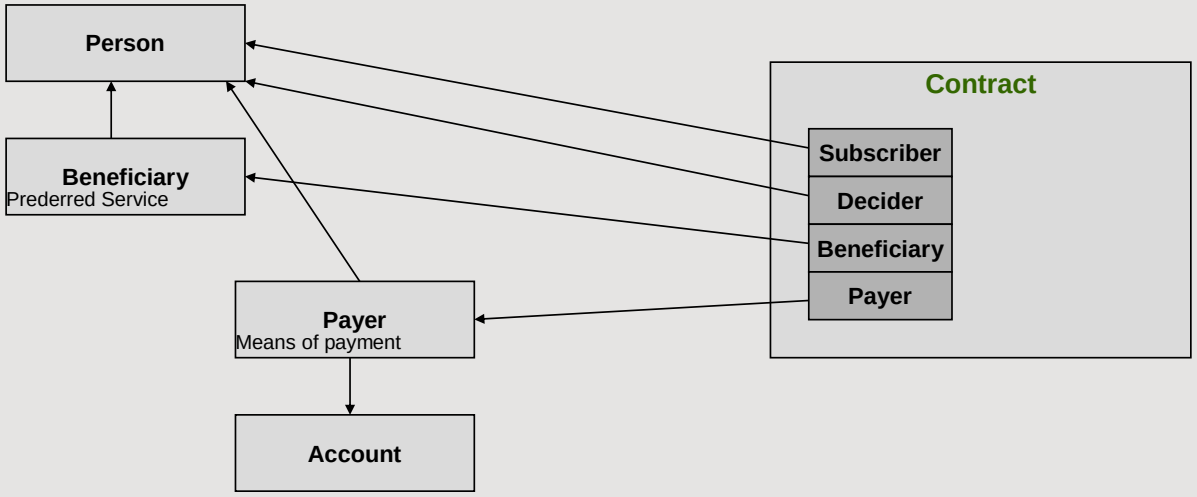
A Relation can be more sophisticated than just defining a link from one Class to another one.
For example,

- a Relation can be single (a Person has only one Address) or **multiple** (a Customer may have several Contracts)
- a Relation can be strong (**Owner**): if a parent Entity is deleted, then the son Entity (like Address of the Person) is also deleted.
    - Remark: a Relation between 2 Entities of 2 different Entity Domains is never an Owner Relation
- a Relation can be **versioned** or not: an Entity relates to the last available version (for example: we just need the last valid address) or a precise version of an Entity (a Claim relates to the version of the Contract when the Claim happened, and not the current version of the Contract)
- **Back** Relation: when A→B relation is defined, when do we define a B→A relation?
    - Generally, we create a Back Relation if from B, it is necessary to call methods from A (which is practically, almost always true)
    - If A→B Is owner : we also automatically create a Back Relation B→A
- **Transaction**: Entities can be strongly linked in a transaction which will group updates on the different Entities

6.6 How to implement a Role?

The same Person may play different Roles: Decider, Subscriber, Beneficiary, Employee, …
Role is implemented as an Attribute or as a Class. Examples:
The Role **Subscriber** is implemented as a single Attribute inside the contract whose name is "Subscriber" and whose Type is 'relation to a Person". We do not need a Class Subscriber, because there is no Attribute or Method which define a Subscriber other than the one already present in the Class Person.
The Role **Employee** is implemented as a Class Employee which as an Attribute whose name is "Person" and whose Type is "relation to a Person". We need the Class Employee because there are specific Employee Attributes like "amount of payroll", "date of first employment in the Company", and specific Rules like "produce an employment certificate".

Implement a Role: Attribute or Class



Page 24

6.7 Prefer several Classes rather then different Perspectives of the same Entity

Some Functions do not require all Attributes of a Entity, they just require a subset of them.
This subset is called "**Perspective**". The same attribute may belong to different Perspectives.
Ex: for the Entity "Person" you may define administrative Perspective (Name, first Name, birth date, place of birth, Social Security number) and sales Perspective (Name, first Name, customer segment, relations with Contracts …).

But implementing the Perspectives is only useful if Classes are big.
Our recommendation is to decompose Entities into several Classes rather than a big Class requiring different perspectives. The advantages are:

- One Design Concept less
- Inheritance is much more powerful on small Classes than on big Classes.

Perspective is a Core Business concept and not an IT concept.

Example:
For the Entity "Person" we suggest to build different Classes like: "General Informations on the Person", "addresses", "Education", "Job", "Assets", "Family", "Behaviour", …

7 Approach to help Business Analysts and IT Developers to reuse the Entity Model

The main principles will be the following:

- Provide an **Entity Model** (definitions, relations, inheritances, identifiers, Life Cycle and Versioning) **before** providing a detailed Data Model.
- Some methodologies make a difference between a **Semantic Model** and an **Entity Design Model**. The Semantic model describes all possible relations between Entities, while the Entity Design Model only describes the ones which have been implemented for a given Enterprise System: For example, the Semantic Model may describe “the Client is related to a Place, and the Place is related to an Address”, while the Entity Design Model only implements “the Client is related to an Address”. The best solution should be: use both models when you design the Entity Model, but **do not communicate the Semantic Model to Business Analysts and IT Developers**: they must have a simple view of what is reusable: the Design Model.
- Propose
  - **Training** on
    - Why it is useful
    - What are the Entities
    - How to create new complementary Entities
  - **Business Analysts coaching** while they try to reuse same Entities when they define requirements.
  - **Developer coaching** while Developers transform requirements into Entity Model.
  - **Certification** of an Entity Model
- Offer a **tool** to
  - **Present** each Entity:
    - Belongs to which Entity Domain (hierarchical)
    - Name
    - Textual definition
    - Example
    - Synonyms
    - Identifier type
    - Status and Life Cycle (State Diagram)
    - Relations with other Entities
    - Inheritance when defined (inheritance will often be defined by the IT Developer and not the Business Analyst)
  - Provide Entity **research** capacity
    - By hierarchy of Entity Domains
    - By Alphabetic research
    - By navigation between Entities
  - Administrate data
    - Define who is responsible for each **Entity definition** or modification
    - Define which level (like “External”, “Group”, “Company”, …) of the Global Enterprise **owns data**
  - Tool options
    - Automatic recognition of other Entity Names in the textual definition to be able to directly access to their definitions
    - Capacity to chose a synonym and global change in all texts

8 Case Study summaries

8.1 AXA case study

AXA Group defined a list of common Entities necessary for the Decision Domain.
They added Life Entities to non Life Entities already available in the IBM model IIA.

8.2 BNP-Paribas case study

This case study explains why BNPP defined an international business glossary.
It also describes the process to define terms: a first version was too ambitious and was reduced to a scope which is used with great success today by the different Companies of the group.
It also explains how these concepts were implemented into the BNPP IT System.

8.3 Michelin case study

These last years, projects were focused on Processes rather than Entities because fashionable methodologies favoured the Process side.
But it was recognized that some difficulties on Enterprise Architecture came from lack of entity definitions.
A set of 1000 Information Objects have been defined at group level.

8.4 Total case study

At project level, Entities have been defined with UML representation.
From project experience, Total is defining a Group Glossary which will be delivered in 3 languages.