# Domain Modeling: Essence and A Sample of Highlights

## (Based partly on [Fowler 1997, Analysis Patterns])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

# Contents


Introduction

Mathematical Preliminaries

Basic Abstractions

Higher Abstractions

Concluding Remarks

# What Is Domain Modeling?



- Domain modeling is an activity of requirements/systems analysis for constructing a conceptual model, usually called the domain model, of the application/problem domain.
- A domain model represents real-world entities/concepts and their relations, to help understand the problem and provide guidelines for software development.
- The focus is often on the data part, though the behavioral aspect is inevitably considered in the modeling process.
- Virtues to pursue: simplicity, flexibility, and reusability.

# Domain Models in UML



- A conceptual/domain model may be described using various modeling notations such as UML class diagrams.
- In a UML class diagram, concepts are represented by classes and relations by relationships, mostly associations and generalizations.

Note: you may want to review the lecture "UML: An Overview" to recall the basics of modeling and UML classes and relationships.

# Identifying Classes, Objects, and Relationships



- Read the problem/requirement statements carefully.
- If there are no such written statements, try to compose them.

# Identifying Classes, Objects, and Relationships

- Read the problem/requirement statements carefully.
- If there are no such written statements, try to compose them.
- Nouns or noun phrases often signal classes/objects.
  - The company has several operating units.

# Identifying Classes, Objects, and Relationships

- Read the problem/requirement statements carefully.
- If there are no such written statements, try to compose them.
- Nouns or noun phrases often signal classes/objects.
  - The company has several operating units.
- Verbs or verb phrases often signal relationships.
  - The company has several operating units.

# Identifying Classes, Objects, and Relationships

- Read the problem/requirement statements carefully.
- If there are no such written statements, try to compose them.
- Nouns or noun phrases often signal classes/objects.
  - The company has several operating units.
- Verbs or verb phrases often signal relationships.
  - The company has several operating units.
- Multiplicity/cardinality is the most fundamental constraint to consider for a relationship.

# Identifying Classes, Objects, and Relationships

- Read the problem/requirement statements carefully.
- If there are no such written statements, try to compose them.
- Nouns or noun phrases often signal classes/objects.
  - The company has several operating units.
- Verbs or verb phrases often signal relationships.
  - The company has several operating units.
- Multiplicity/cardinality is the most fundamental constraint to consider for a relationship.
- Constraints that cannot be easily captured by multiplicities may be stated in a note. (Can you think of one such constraint?)

# Multiplicities of Relationships

- The usual classification:
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many

  (This classification is in fact incomplete and should be refined.)

# Multiplicities of Relationships

- The usual classification:
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many

  (This classification is in fact incomplete and should be refined.)
- A relationship can be conveniently modeled as a (mathematical) binary relation, which has a direction.

# Multiplicities of Relationships

- The usual classification:
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many

  (This classification is in fact incomplete and should be refined.)
- A relationship can be conveniently modeled as a (mathematical) binary relation, which has a direction.
- So, many-to-one and one-to-many relationships/relations should be treated differently.

# Multiplicities of Relationships

- The usual classification:
  - one-to-one
  - many-to-one
  - one-to-many
  - many-to-many

  (This classification is in fact incomplete and should be refined.)
- A relationship can be conveniently modeled as a (mathematical) binary relation, which has a direction.
- So, many-to-one and one-to-many relationships/relations should be treated differently.
- One should also be careful about from which side of a "one-to-one" relationship the relation is a total function.

# Sets and Types

IM NTU

- A *set* is a collection of things/objects, each called an *element* of the set.
- A set may be built from existing sets:
  - Union, intersection, and complement
  - Subset and power set
  - Product

# Sets and Types

- A *set* is a collection of things/objects, each called an *element* of the set.
- A set may be built from existing sets:
  - Union, intersection, and complement
  - Subset and power set
  - Product
- A *multiset* allows repetitions of a same element; use this notion when the ordinary set is not suitable.

# Sets and Types

- A *set* is a collection of things/objects, each called an *element* of the set.
- A set may be built from existing sets:
  - Union, intersection, and complement
  - Subset and power set
  - Product
- A *multiset* allows repetitions of a same element; use this notion when the ordinary set is not suitable.
- One can think of an element $a$ from a set $A$ as being of *type* $A$.
- So, types or data types basically are just sets; and subtypes are subsets. (More about this later.)

# Tuples and Records

IM NTU

- A *tuple* is a finite ordered list (sequence) of elements, each called a *component* of the tuple, such as ("IM5027", "Software Development Methods", 3).
- The components of a tuple may be of different types.

# Tuples and Records



- A *tuple* is a finite ordered list (sequence) of elements, each called a *component* of the tuple, such as (“IM5027”, “Software Development Methods”, 3).
- The components of a tuple may be of different types.
- A tuple with $k$ ($k \geq 0$) components is called a *k-tuple*.
- A 2-tuple is usually called a *pair*.

# Tuples and Records

- A *tuple* is a finite ordered list (sequence) of elements, each called a *component* of the tuple, such as ("IM5027", "Software Development Methods", 3).
- The components of a tuple may be of different types.
- A tuple with $k$ ($k \geq 0$) components is called a *$k$-tuple*.
- A 2-tuple is usually called a *pair*.
- The *Cartesian product*, or simply product, of $A$ and $B$, written as $A \times B$, is the set of all pairs $(x, y)$ such that $x \in A$ and $y \in B$.
- For example, $\{a, b\} \times \{0, 1, 2\} = \{(a, 0), (a, 1), (a, 2), (b, 0), (b, 1), (b, 2)\}$.

# Tuples and Records (cont.)

- Cartesian products generalize to $k$ sets, $A_1$, $A_2$, ..., $A_k$, written as $A_1 \times A_2 \times \ldots \times A_k$.
- So, every element of $A_1 \times A_2 \times \ldots \times A_k$ is a $k$-tuple.
- $A^k$ is a shorthand for $A \times A \times \ldots \times A$ ($k$ times).

# Tuples and Records (cont.)

- Cartesian products generalize to $k$ sets, $A_1, A_2, \ldots, A_k$, written as $A_1 \times A_2 \times \ldots \times A_k$.
- So, every element of $A_1 \times A_2 \times \ldots \times A_k$ is a $k$-tuple.
- $A^k$ is a shorthand for $A \times A \times \ldots \times A$ ($k$ times).
- A *record* is essentially a generalization of a tuple, where every component is given a name, called a *field name* or *attribute*.
- Below is an example record:
  (ID: "IM5027", Title: "Software Development Methods", Credit: 3).

# Relations

- A subset $R$ of $A_1 \times A_2 \times \ldots \times A_k$ is called a $k$-ary *relation* on $A_1, A_2, \ldots, A_k$.
- We usually write $R(a_1, a_2, \ldots, a_k)$ to denote that $(a_1, a_2, \ldots, a_k) \in R$.
- So, one can view a relation $R \subseteq A_1 \times A_2 \times \ldots \times A_k$ as a *predicate*.
- When the $A_i$'s are the same set $A$, it is simply called a $k$-ary relation on $A$.

# Relations (cont.)



- A 1-ary relation is usually called a *unary relation*, which is also a way of defining subsets from an existing set.
- A 2-ary relation is called a *binary relation*; for a binary relation $R$, $R(x, y)$ is also written as $xRy$.
- A binary relation $R \subseteq A \times B$ is said to be *total* if, for every $x \in A$, there exists some $y \in B$ such that $xRy$.
- Binary relations are the most used relations.

# Relations (cont.)

- A 1-ary relation is usually called a *unary relation*, which is also a way of defining subsets from an existing set.
- A 2-ary relation is called a *binary relation*; for a binary relation $R$, $R(x, y)$ is also written as $xRy$.
- A binary relation $R \subseteq A \times B$ is said to be *total* if, for every $x \in A$, there exists some $y \in B$ such that $xRy$.
- Binary relations are the most used relations.
- $R \subseteq (A_1 \times A_2 \times \ldots \times A_m) \times (B_1 \times B_2 \times \ldots \times B_n)$ is a binary relation on $A_1 \times A_2 \times \ldots \times A_m$ and $B_1 \times B_2 \times \ldots \times B_n$.

# Relations (cont.)

- A 1-ary relation is usually called a *unary relation*, which is also a way of defining subsets from an existing set.
- A 2-ary relation is called a *binary relation*; for a binary relation $R$, $R(x, y)$ is also written as $xRy$.
- A binary relation $R \subseteq A \times B$ is said to be *total* if, for every $x \in A$, there exists some $y \in B$ such that $xRy$.
- Binary relations are the most used relations.
- $R \subseteq (A_1 \times A_2 \times \ldots \times A_m) \times (B_1 \times B_2 \times \ldots \times B_n)$ is a binary relation on $A_1 \times A_2 \times \ldots \times A_m$ and $B_1 \times B_2 \times \ldots \times B_n$.
- Alternatively, $R \subseteq A_1 \times A_2 \times \ldots \times A_m \times B_1 \times B_2 \times \ldots \times B_n$ is a $(m + n)$-ary relation.

# Functions

- A (total) *function* (or mapping) $f$ from $D$ to $R$, denoted $f : D \longrightarrow R$, maps every element in $D$, called the *domain* of $f$, to some element in $R$, called the *range* of $f$.
- A function sets up an *input-output* relationship between its domain and range, where the same input always produces the same output.

# Functions

- A (total) *function* (or mapping) $f$ from $D$ to $R$, denoted $f : D \longrightarrow R$, maps every element in $D$, called the *domain* of $f$, to some element in $R$, called the *range* of $f$.
- A function sets up an *input-output* relationship between its domain and range, where the same input always produces the same output.
- A function $f : D \longrightarrow R$ may be seen as a special kind of total binary relation $f \subseteq D \times R$ that is *functional* (one-to-one or many-to-one), i.e., for every $d \in D$, there is exactly an $r \in R$ s.t. $(d, r) \in f$, written usually as $f(d) = r$.

# Functions

- A (total) *function* (or mapping) $f$ from $D$ to $R$, denoted $f : D \longrightarrow R$, maps every element in $D$, called the *domain* of $f$, to some element in $R$, called the *range* of $f$.
- A function sets up an *input-output* relationship between its domain and range, where the same input always produces the same output.
- A function $f : D \longrightarrow R$ may be seen as a special kind of total binary relation $f \subseteq D \times R$ that is *functional* (one-to-one or many-to-one), i.e., for every $d \in D$, there is exactly an $r \in R$ s.t. $(d, r) \in f$, written usually as $f(d) = r$.
- A *partial* function may not produce an output for some inputs.

# Functions (cont.)

- A function is said to be *$k$-ary* if its domain is a product of $k$ sets.
- That is, $f : D_1 \times D_2 \times \ldots \times D_k \longrightarrow R$ is called a $k$-ary function.

# Functions (cont.)

- A function is said to be *k-ary* if its domain is a product of $k$ sets.
- That is, $f : D_1 \times D_2 \times \ldots \times D_k \longrightarrow R$ is called a $k$-ary function.
- Recall that $f$ may be seen as a special kind of binary relation, i.e., $f \subseteq (D_1 \times D_2 \times \ldots \times D_k) \times R$.

# Functions (cont.)

- A function is said to be *$k$-ary* if its domain is a product of $k$ sets.
- That is, $f : D_1 \times D_2 \times \ldots \times D_k \longrightarrow R$ is called a $k$-ary function.
- Recall that $f$ may be seen as a special kind of binary relation, i.e., $f \subseteq (D_1 \times D_2 \times \ldots \times D_k) \times R$.
- Function $f$ may also be seen as a special kind of $(k+1)$-ary relation, i.e., $f \subseteq D_1 \times D_2 \times \ldots \times D_k \times R$.

# “Multi-Valued Functions”

- When a binary relation $f \subseteq D \times R$ is total but not functional, it is sometimes referred to as a “multi-valued function”.

# "Multi-Valued Functions"

- When a binary relation $f \subseteq D \times R$ is total but not functional, it is sometimes referred to as a "multi-valued function".
- Can we represent it as a real (single-valued) function?

# "Multi-Valued Functions"

- When a binary relation $f \subseteq D \times R$ is total but not functional, it is sometimes referred to as a "multi-valued function".
- Can we represent it as a real (single-valued) function?
- Define a function $g : D \longrightarrow 2^R$ (from $D$ to the power set of $R$) such that, for every $d \in D$, $g(d) = \{r \in R \mid (d, r) \in f\}$.
- Function $g$ is single-valued and faithfully represents $f$.

# Subsets, Subtypes, and Subclasses

IM NTU

- How can subtypes, or even subclasses, simply be viewed as subsets?
  (Classes have their behavioral aspects, but that does not concern us in this lecture, which focuses on the data part).

# Subsets, Subtypes, and Subclasses

- How can subtypes, or even subclasses, simply be viewed as subsets?
  (Classes have their behavioral aspects, but that does not concern us in this lecture, which focuses on the data part).
- Doesn't an object of a subtype/subclass has more attributes?

# Subsets, Subtypes, and Subclasses

- How can subtypes, or even subclasses, simply be viewed as subsets?
  (Classes have their behavioral aspects, but that does not concern us in this lecture, which focuses on the data part).
- Doesn't an object of a subtype/subclass has more attributes?
- Mathematical relations can be conveniently used to represent types/classes and are themselves sets.
- A $k$-ary relation, when seen as a predicate, constrains its $k$ components and nothing beyond.

# Subsets, Subtypes, and Subclasses

IM NTU

- How can subtypes, or even subclasses, simply be viewed as subsets?
  (Classes have their behavioral aspects, but that does not concern us in this lecture, which focuses on the data part).
- Doesn't an object of a subtype/subclass has more attributes?
- Mathematical relations can be conveniently used to represent types/classes and are themselves sets.
- A $k$-ary relation, when seen as a predicate, constrains its $k$ components and nothing beyond.
- A $k$-tuple $(d_1, d_2, \ldots, d_k)$ in a $k$-ary relation may be extended as a $(k+1)$-tuple $(d_1, d_2, \ldots, d_k, \_)$, where the $(k+1)$-th component may contain any value ("don't care"), denoted by $\_$.
- The extension may be generalized to include more than one additional components.

# Why Mathematics?

- It is precise.
  - Being abstract/conceptual does not imply being vague/imprecise.
  - Abstraction is about singling out commonalities and removing/hiding unnecessary details.

# Why Mathematics?

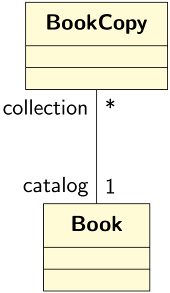
- It is precise.
  - Being abstract/conceptual does not imply being vague/imprecise.
  - Abstraction is about singling out commonalities and removing/hiding unnecessary details.
- It is common ultimately, for all.

# Why Mathematics?

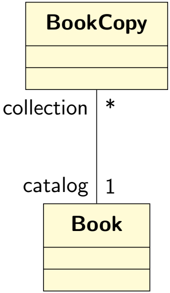
IM NTU

- It is precise.
  - Being abstract/conceptual does not imply being vague/imprecise.
  - Abstraction is about singling out commonalities and removing/hiding unnecessary details.
- It is common ultimately, for all.
- It is expressive.

# Books vs. Book Copies

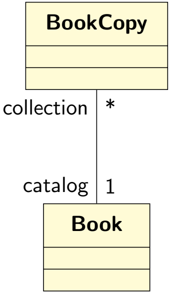


# Books vs. Book Copies




It is a BookCopy, not a Book, that you reserve or check out from the library!

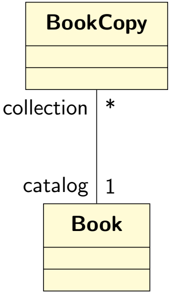
# Books vs. Book Copies




It is a BookCopy, not a Book, that you reserve or check out from the library!

Can you see an association relationship as a binary relation mathematically?

# Books vs. Book Copies




It is a BookCopy, not a Book, that you reserve or check out from the library!

Can you see an association relationship as a binary relation mathematically?

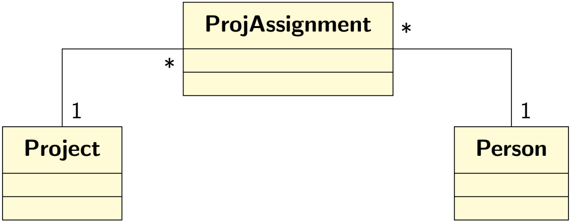
Can you see in an association two functions mathematically?

# An Association Class


A many-to-many relation (at the operational level) should be avoided. Why?

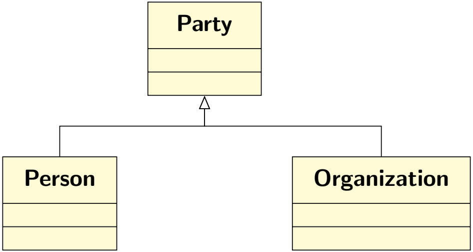
# An Association Class


A many-to-many relation (at the operational level) should be avoided. Why? It may instead be represented as follows.



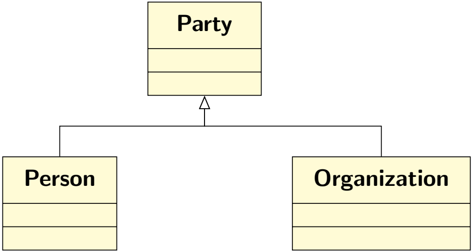

The class ProjAssignment is called an association class, created to represent the original many-to-many association relation between Project and Person.

# The Abstract Concept/Class of "Party"




The Party generalization may apply to other entities, e.g., Post.

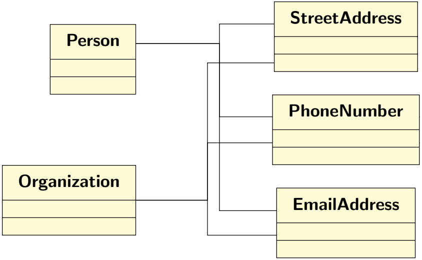
# The Abstract Concept/Class of “Party”

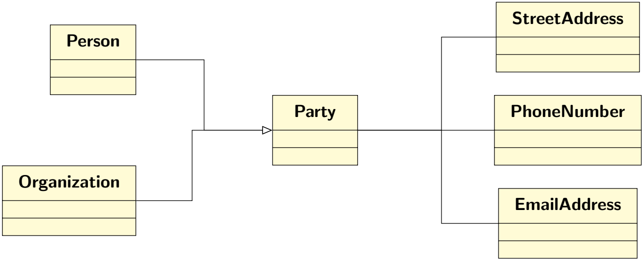



The Party generalization may apply to other entities, e.g., Post.

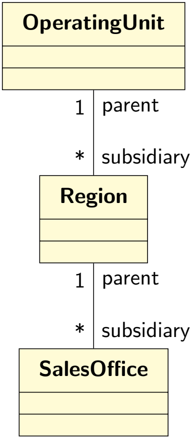
Can you see Person and Organization as subsets of Party mathematically?

# The Party Abstraction Simplifies Relations

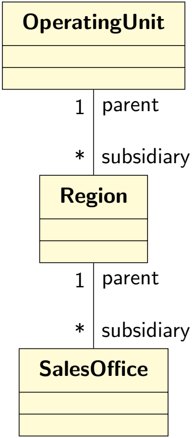


# Hierarchies

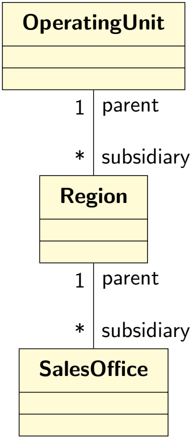


# Hierarchies




What if additional levels are needed?

# Hierarchies

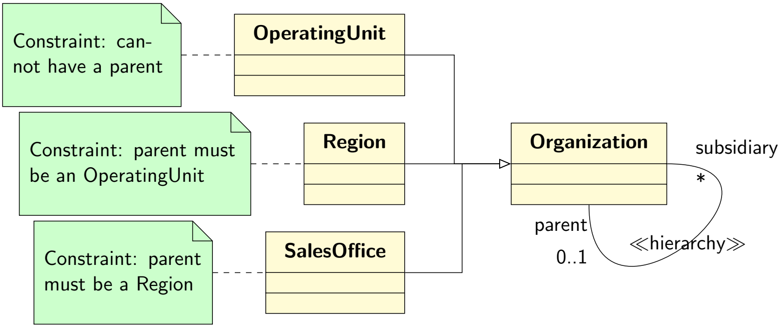



What if additional levels are needed?

Modeling a hierarchy with explicit levels is inflexible.

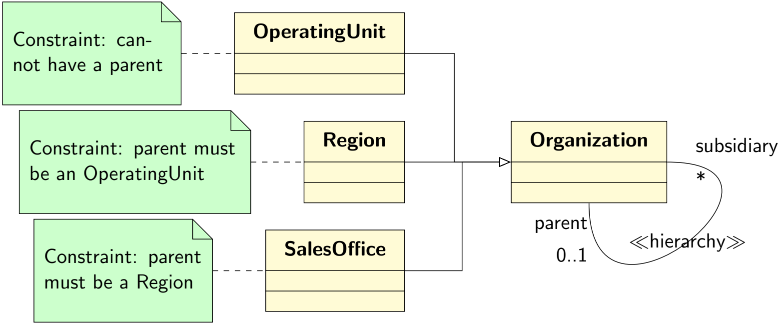
# Hierarchies (cont.)

A hierarchical association provides better flexibility:



# Hierarchies (cont.)

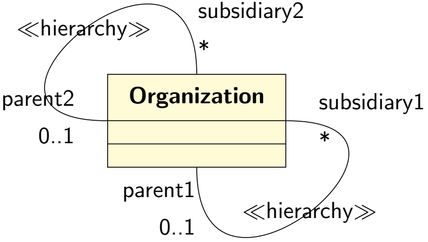
A hierarchical association provides better flexibility:




Can you see the hierarchical association as a binary relation on Organization mathematically?

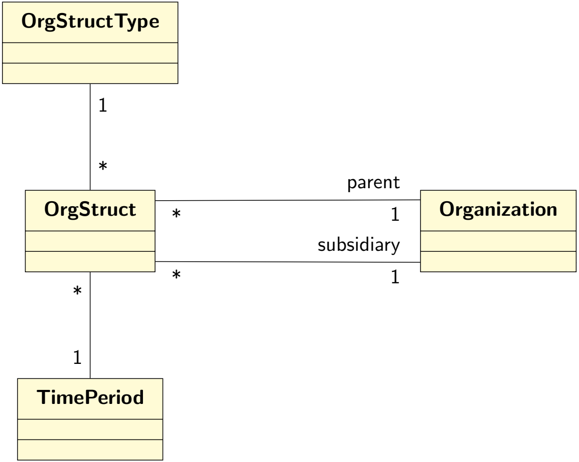
# More about Hierarchies

What if several different hierarchies are needed?

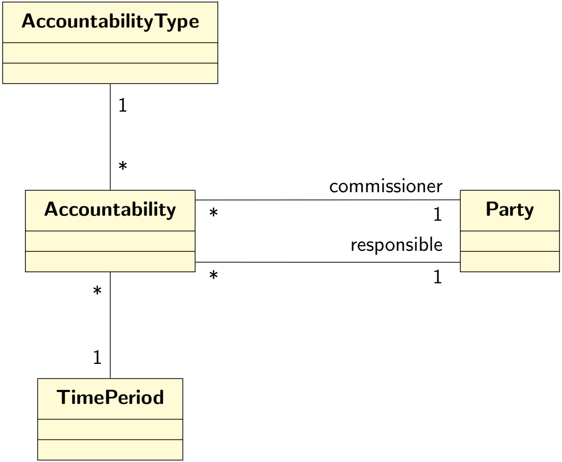



This will become messy, when there are many hierarchies.

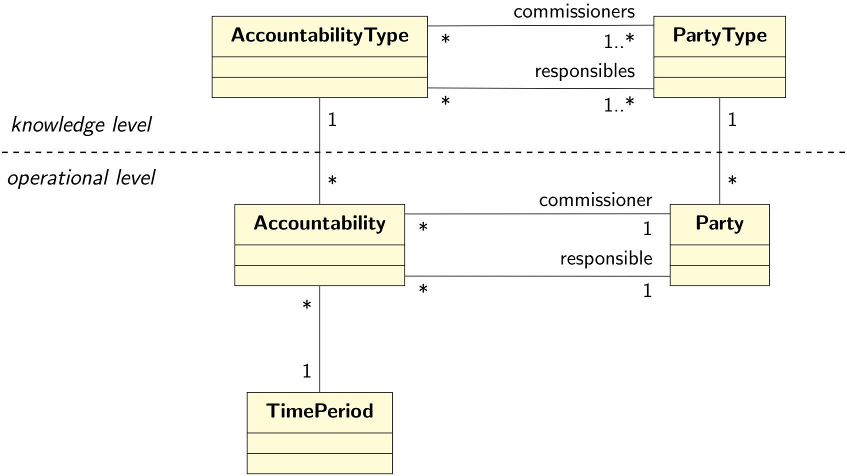
# Typed Relationship



# Accountability



# Knowledge vs. Operational Levels



# Concluding Remarks

- Domain modeling requires domain knowledge and experience.
- Experience can be passed on and learned by good examples, namely patterns.
- Patterns are not fixed and should be adapted to fit your needs.
- Always strive for simplicity, flexibility, and reusability.