

# Domain Modeling: A Brief Introduction

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

Yih-Kuen Tsay

Department of Information Management  
National Taiwan University

# Contents

Introduction

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.

- 🌐 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.
- 🌐 The classes and relationships may be implicit in the statements.

- 🌐 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.
- 🌐 The classes and relationships may be implicit in the statements.
- 🌐 What about the multiplicities of a relationship?

- 🌐 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.
- 🌐 The classes and relationships may be implicit in the statements.
- 🌐 What about the multiplicities of a relationship?
- 🌐 Constraints that cannot be easily captured by multiplicities may be stated in a note.

# 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
- 🌐 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

- 🌐 A *tuple* is a finite ordered list (sequence) of elements, each called a *component* of the tuple.
- 🌐 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  $y \in B$ .
- 🌐 Cartesian products generalize to  $k$  sets,  $A_1, A_2, \dots, A_k$ , written as  $A_1 \times A_2 \times \dots \times A_k$ .
- 🌐 So, every element of  $A_1 \times A_2 \times \dots \times A_k$  is a *k-tuple*.
- 🌐  $A^k$  is a shorthand for  $A \times A \times \dots \times A$  ( $k$  times).
- 🌐 A *record* is essentially a generalization of a tuple, where every component is given a name, called a *field name* or *attribute*.

# Relations and Predicates

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

# Functions

- 🌐 A *function* (or mapping)  $f$  from  $D$  to  $R$ , denoted  $f : D \rightarrow R$ , 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.
- 🌐 So, a function  $f : D \rightarrow R$  may be seen as a special kind of binary relation  $f \subseteq D \times R$  that is *functional* (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 function is said to be *k-ary* if its domain is a product of  $k$  sets.
- 🌐 A *k-ary* function  $f : D_1 \times D_2 \times \dots \times D_k \rightarrow R$  may also be seen as a special kind of  $(k + 1)$ -ary relation  $f \subseteq D_1 \times D_2 \times \dots \times D_k \times R$ .
- 🌐 A *partial* function may not produce an output for some inputs.

# Subsets, Subtypes, and Subclasses

- 🌐 How can subtypes/subclasses be simply viewed as subsets?
- 🌐 Doesn't an object of a subclass has more attributes?

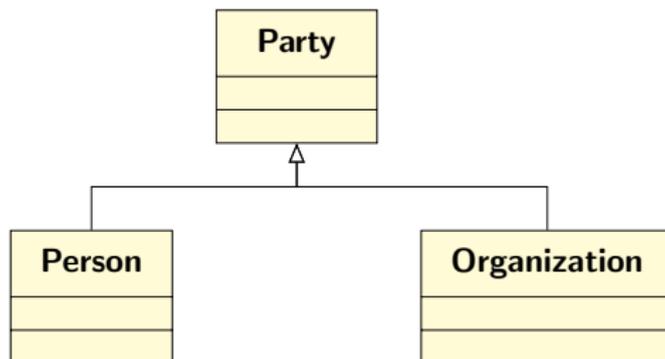
# Subsets, Subtypes, and Subclasses

- How can subtypes/subclasses be simply viewed as subsets?
- Doesn't an object of a subclass has more attributes?
- Relations (mathematical relations) are themselves sets and can be used to represent classes.
- A  $k$ -ary relation, when seen as a predicate, constrains its  $k$  components and **nothing beyond**.
- A  $k$ -tuple  $(d_1, d_2, \dots, d_k)$  in a  $k$ -ary relation may be extended as a  $(k + 1)$ -tuple  $(d_1, d_2, \dots, d_k, \_)$ , 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?

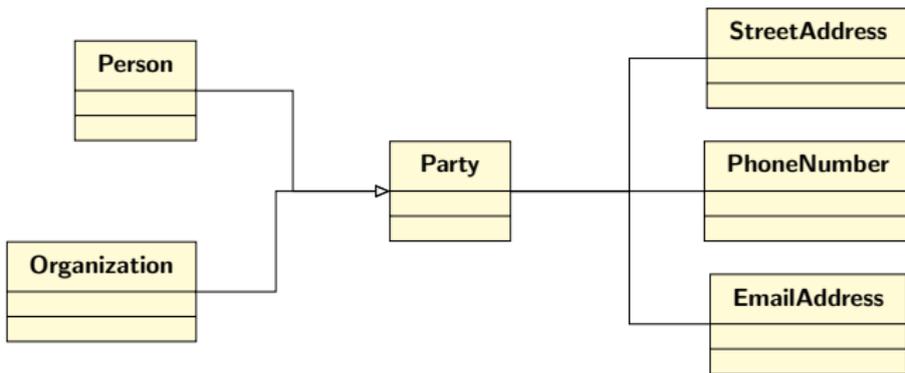
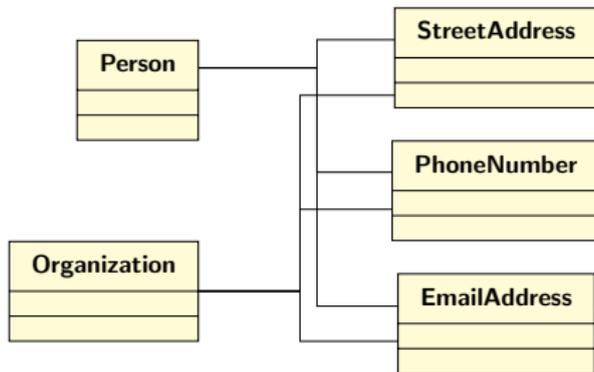
- 🌐 It is precise.
  - ☀️ Being abstract/conceptual does not imply being vague/imprecise.
  - ☀️ Abstraction is about singling out commonalities and removing/hiding unnecessary details.
- 🌐 It is common, for all.
- 🌐 It is expressive.

# The Abstract Concept/Class of “Party”



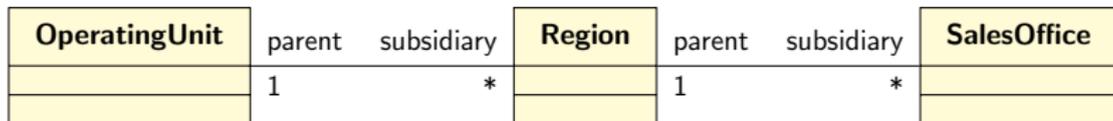
The Party generalization may apply to other entities, e.g., Post.

# The Party Abstraction Simplifies Relations

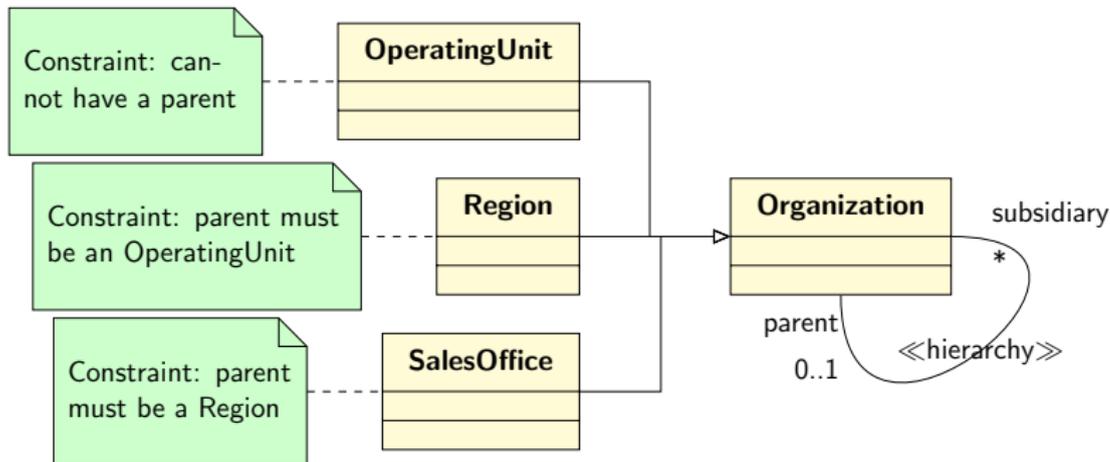


# Hierarchies

Explicit levels are inflexible:



A hierarchical association provides better flexibility:



# An Association Class

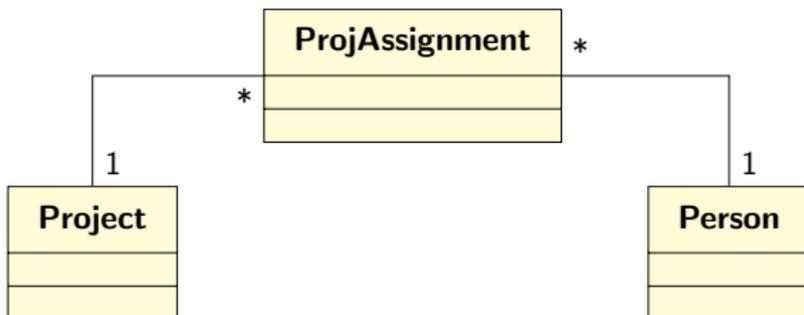


A many-to-many relation (at the operational level) should be avoided. Why?

# An Association Class

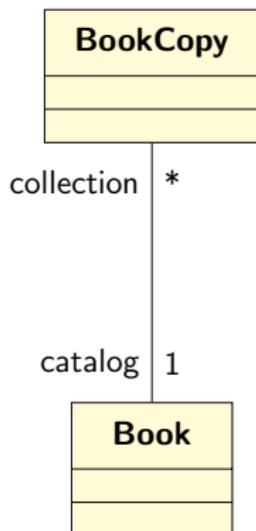


A many-to-many relation (at the operational level) should be avoided. Why? It may instead be represented as follows.



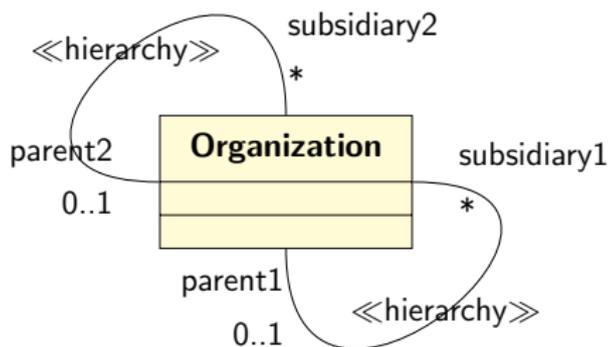
The class **ProjAssignment** is called an **association class**, created to represent the original many-to-many association relation between **Project** and **Person**.

# Books vs. Book Copies

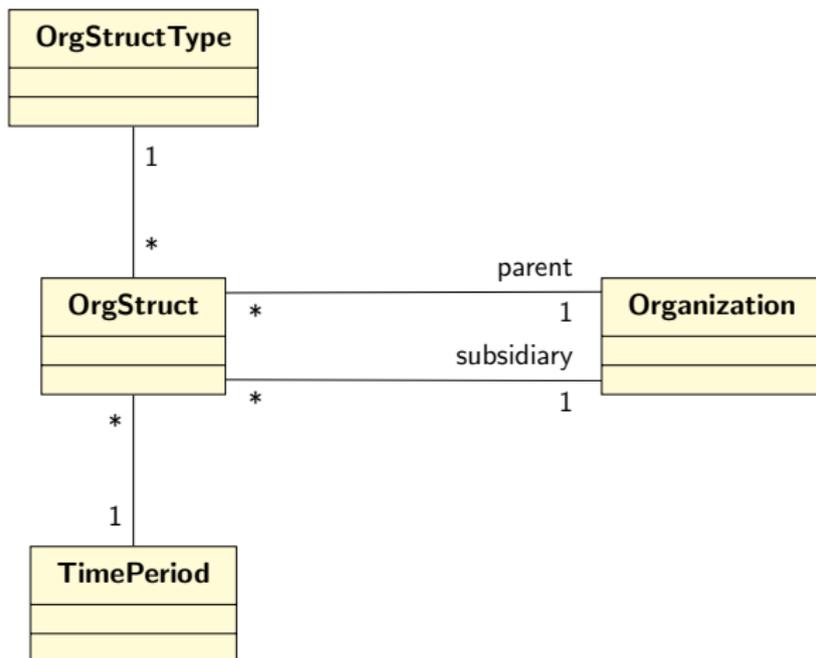


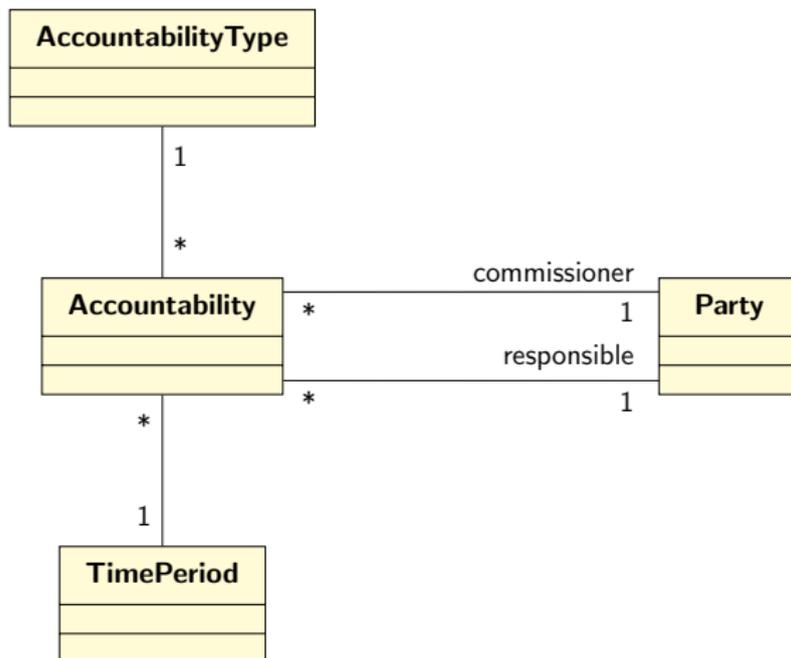
# More about Hierarchies

What if several different hierarchies are needed?

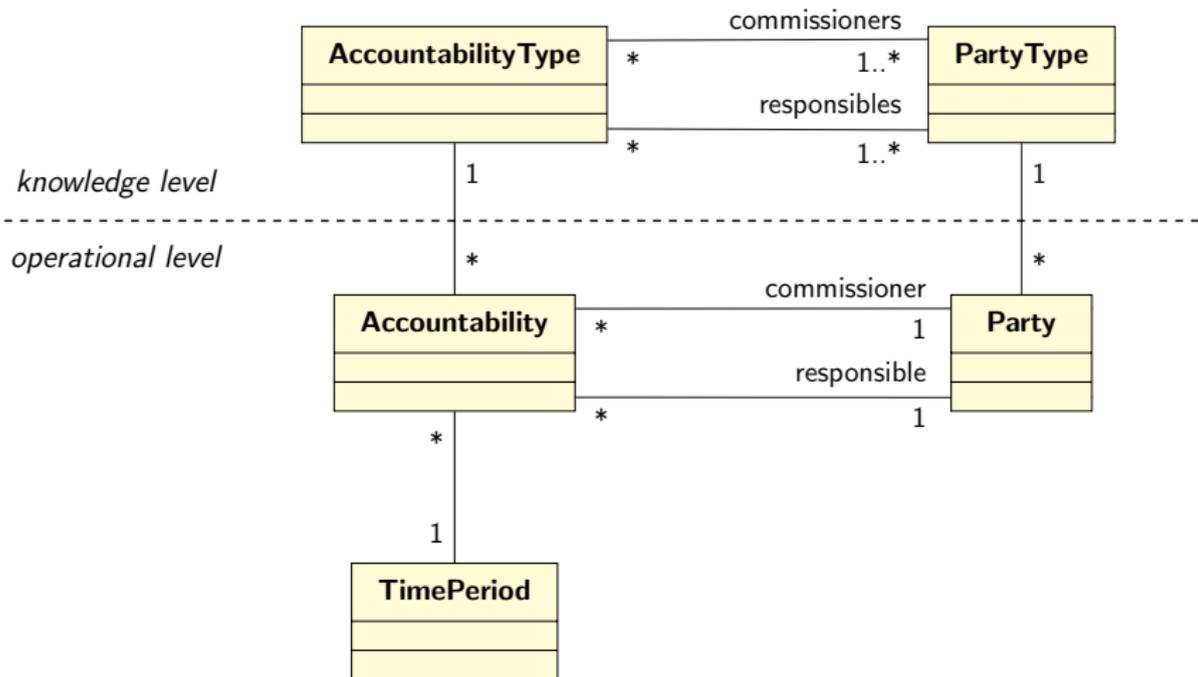


# Typed Relationship





# 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.