Topic 4.3: Data Structure Patterns

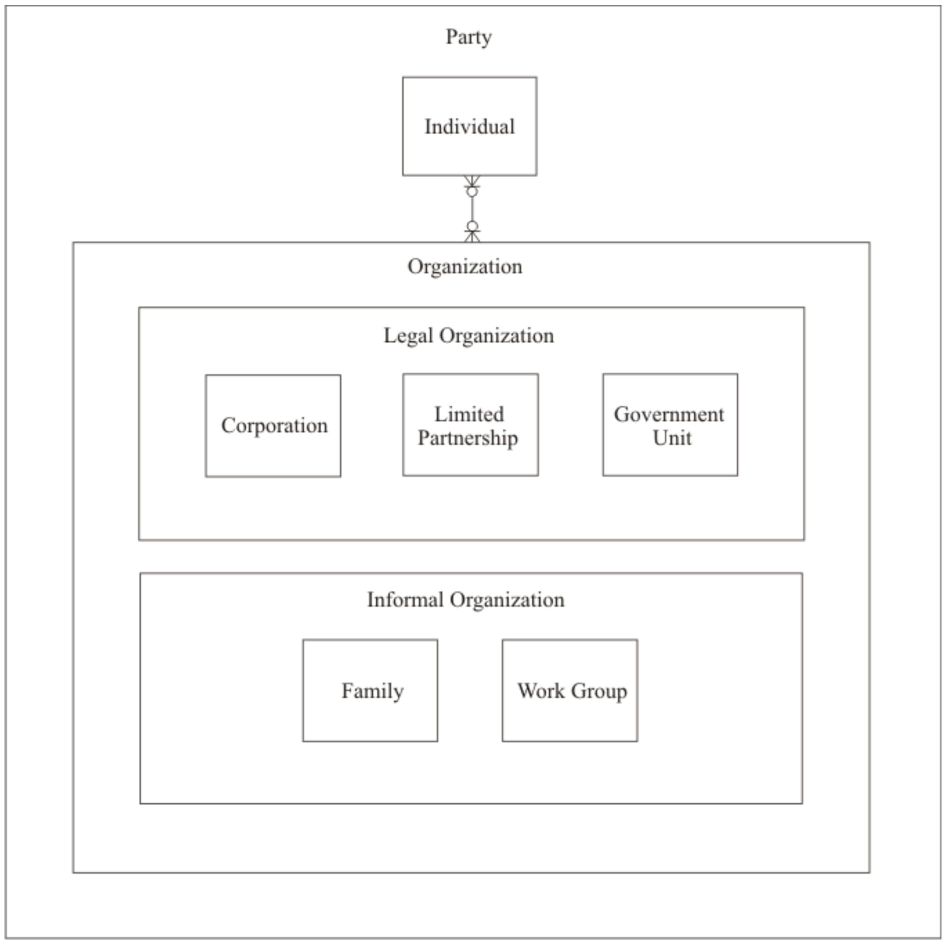
Very few data modeling projects require you to start with the proverbial blank slate. Seek out a conceptual model for a similar application or database that has worked either for you or for the organization. If none is found, look for common data structures and patterns that you or the organization have used before. Over time, you will develop your own library of models and common structures available for reuse. Most modelers start with some sort of generic model or pattern even if they don't consciously realize it.

A number of publications and web pages can provide useful generic models and patterns. One of the very best sources is the three-volume set *The Data Model Resource Book,* Volumes 1 and 2 by Len Silverston (Wiley Publishing, 2001), and Volume 3 by Len Silverston and Paul Agnew (Wiley Publishing, 2009). Silverston has long touted the concept of a universal data model. Even if you cannot adapt generic models directly, they provide a very attractive starting point. The key is to *adapt* rather than to force a model fit.

The following topics present the Party, Role and Generalized Hierarchy data structures as examples of commonly used pattern data structures.

The Party Model

The following diagram shows a pattern model for the super type commonly called Party. A *party* is a person or organization of interest to the database application. The advantage of using the super type in data models is the ability to hide the party's implementation details in cases where it doesn't matter. For example, in a workflow application some tasks are assigned to individual specialists, while others are assigned to workgroups. If we were to use separate entities for an individual and a workgroup, we'd have to have two relationships, one between tasks and individuals and another between tasks and workgroups. And if that weren't complicated enough, we'd have to know the type of task in order to write a database query that links a task to the person or workgroup to which it was assigned. On the other hand, using the Party super type, we need only a single relationship and assign tasks to parties without regard for whether the party was an individual or organization.



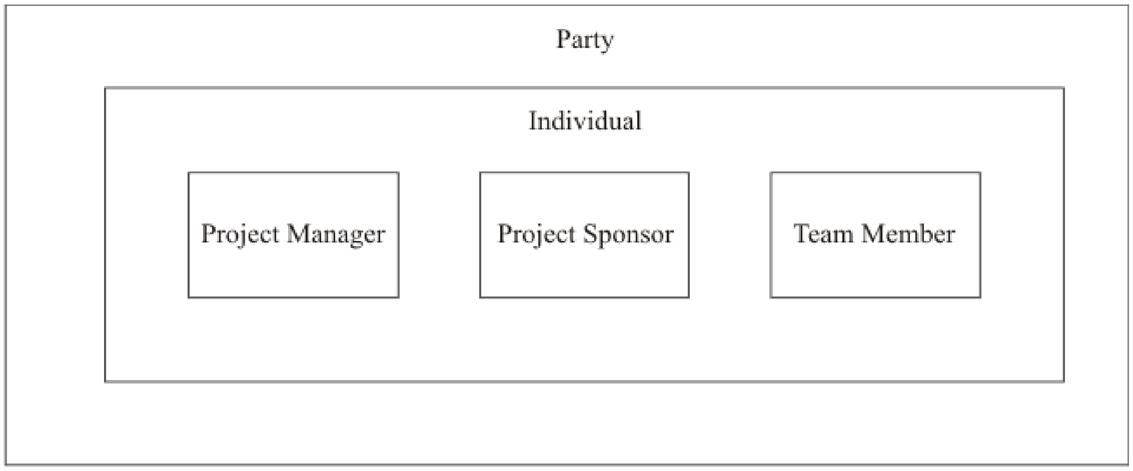
Note that supertypes do not have to be uniformly broken into subtypes. For example, in the preceding diagram, Party is broken down into Individual and Organization, and while Individual isn't broken down further, Organization is broken down into Legal Organization and Informal Organization, and each of those is also broken into subtypes. Obviously, the subtypes need to be adjusted to fit the organization and the particular requirements of the project. Said another way, you should never generalize or specialize data structures merely for the sake of doing so—your design decisions should *always* be driven by your *requirements.*

The Party structure shown in the diagram has an unusual element: the subtype Individual has a many-to-many relationship with the subtype Organization. An individual can be a member of a family and a workgroup, for example; and it should be obvious that a family or workgroup can have many individuals as members. In the next topic, I introduce roles as another option for implementing relationships such as this one.

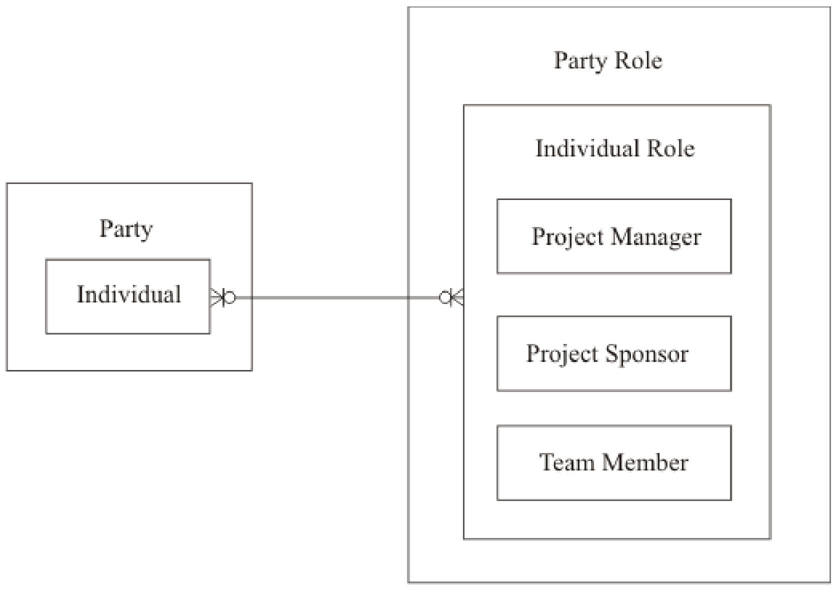
The Role Model

The role model is an alternative to using super types and subtypes. When we break entities down into subtypes, there are two guiding principles. First, the subtypes must not overlap, meaning any given occurrence of the entity must be exactly one subtype. For example, you won't often see customer and supplier as subtypes of organization because there will likely be cases where the same organization is both a customer and supplier such as the electrical contractor that buys a truck from the vehicle dealer for which they perform electrical work. Second, subtypes must cover all the possible cases, meaning that when an entity is broken into subtypes, every occurrence must fit one of the subtypes. If, for example, an organization deals with regulatory agencies, the agencies are neither customers nor suppliers, so we would have to make a Regulatory Agency subtype in the previous example. Putting the two principles together, if we break an entity into subtypes, then every occurrence of the entity must map to one and only one subtype. (Note that not all practitioners agree with these two principles.)

The following diagram shows a different version of the Party entity shown in the previous diagram, with the Individual subtype broken down into Project Manager, Project Sponsor, and Team Member subtypes. Even if an individual would never serve in more than one of these functions at the same time, imagine how awkward it would be every time a person changed from one to the other—if each subtype were implemented as a separate table, we would be forever moving rows from one table to another as individuals changed job functions within the organization.

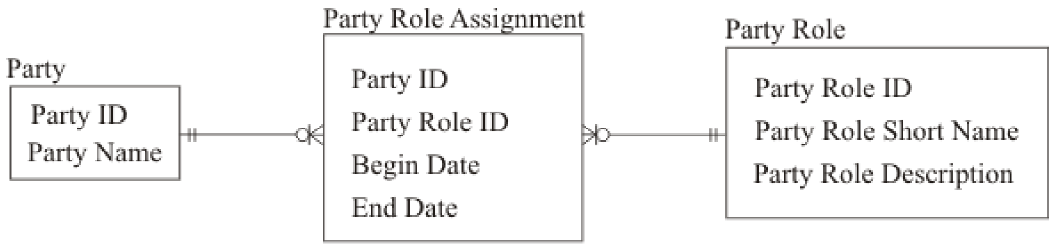


In situations where it is impossible or impractical to design subtypes that conform to the guiding principles previously described, the solution is to use roles. In data modeling, a *role* represents the part that a particular entity class might play in the organization, much like a role played by an actor on stage or screen. The following shows a conceptual model where Project Manager, Project Sponsor, and Team Member are treated as subtypes of Individual Role, which in turn is a subtype of Party Role. The assignment of an Individual to a role is now done with a relationship between the Individual entity and the Individual Role entity, which allows the assignment of more than one role at a given point in time if needed.

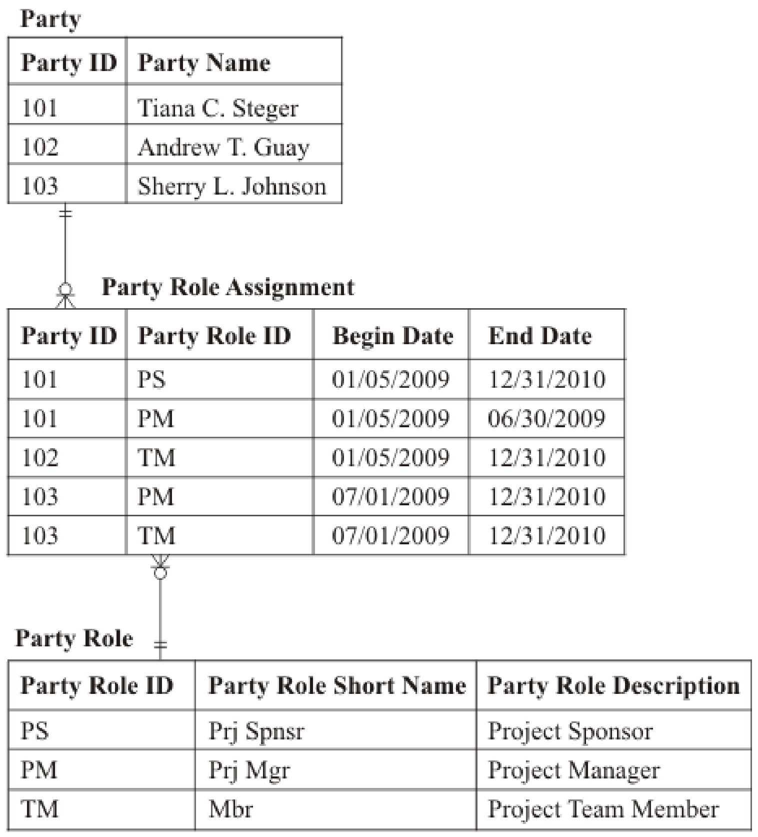


In reality, we would never implement such a complicated super type/subtype structure as shown in the Party Role entity shown in the diagram; but as a conceptual model, it has the benefit of graphically showing that an Individual can be assigned only roles defined as Individual Roles, which are Project Manager, Project Sponsor, or Team Member. Perhaps the most significant reason we would not actually implement this structure in a logical or physical model is that we would have to modify the data model (and perhaps the database) every time a new role type became necessary, which is obviously too inflexible for most organizations.

The following diagram shows the most likely logical model derived from the conceptual model shown in the previous diagram. Note that I have flattened the Party Role structure into a single table that contains all possible roles a party can play and have placed an intersection entity between Party and Party Role to show the assignment of parties to party roles. (The intersection entity also resolves the many-to-many relationship between Party and Party Role to comply with normalization rules. However, normalization is not necessary in conceptual models.) I added begin and end effective dates to the intersection entity (Party Role Assignment) to allow past and future role assignments to exist along with current assignments.



Note that a Party can have zero to many related Party Role Assignment instances, and similarly, a Party Role can have zero to many related Party Role Assignment instances. To help you visualize how this model works, the following diagram shows three physical tables (a rather rudimentary form of physical data model) with sample data. The three possible individual roles from the conceptual model (Project Manager, Project Sponsor, and Team Member) now appear as rows in the Party Role table. Three parties are shown in the Party table, and the Party Role Assignment table shows various past and present role assignments. In particular, note that Tiana C. Steger (Party 101) served as both a Project Sponsor and a Project Manager between January 5, 2009, and June 30, 2009. Such a dual assignment would be against the rules had we used subtypes instead of roles.



Party

Party ID	Party Name
101	Tiana C. Steger
102	Andrew T. Guay
103	Sherry L. Johnson

Party Role Assignment

Party ID	Party Role ID	Begin Date	End Date
101	PS	01/05/2009	12/31/2010
101	PM	01/05/2009	06/30/2009
102	TM	01/05/2009	12/31/2010
103	PM	07/01/2009	12/31/2010
103	TM	07/01/2009	12/31/2010

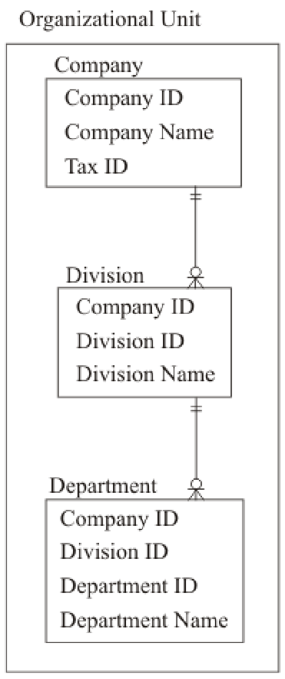
Party Role

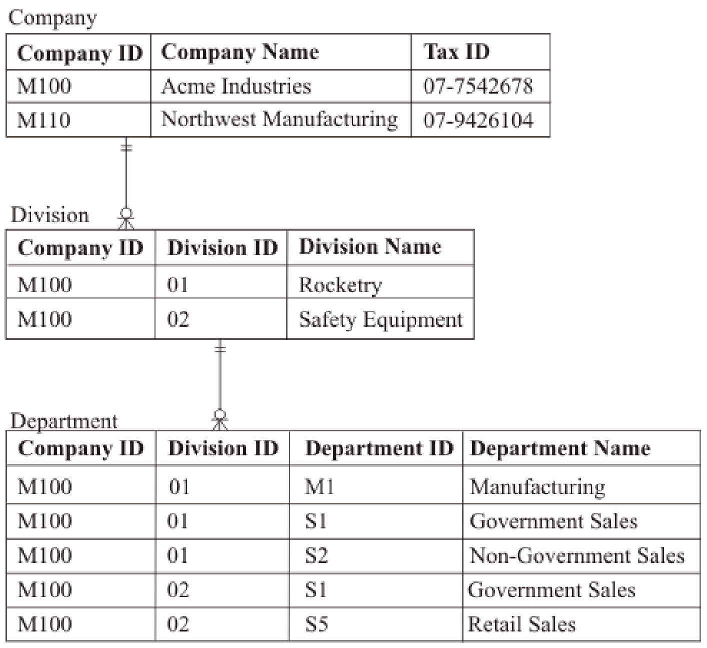
Party Role ID	Party Role Short Name	Party Role Description
PS	Prj Spnsr	Project Sponsor
PM	Prj Mgr	Project Manager
TM	Mbr	Project Team Member

The Generalized Hierarchy Model

Hierarchies are structures where entities are organized into layers in which each entity can have any number of children (subordinate entities) at lower layers, but can have only one parent (superior entity) at the next higher layer. And, of course, there is a single entity in the top layer that has no parents with respect to the hierarchy, and entities at the bottom of the hierarchy have no children. Hierarchies are sometimes called tree structures because they look like an upside-down tree, with the topmost entity called the root and the bottommost entities called leaves.

The following diagram shows a familiar three-layer hierarchy for an enterprise organization chart containing the entities Company, Division, and Department. A conceptual model with attributes is shown at the top of the diagram, and a physical representation with sample data values is shown at the bottom of the diagram.





Company

Company ID	Company Name	Tax ID
M100	Acme Industries	07-7542678
M110	Northwest Manufacturing	07-9426104

Division

Company ID	Division ID	Division Name
M100	01	Rocketry
M100	02	Safety Equipment

Department

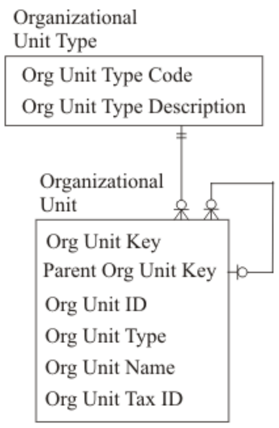
Company ID	Division ID	Department ID	Department Name
M100	01	M1	Manufacturing
M100	01	S1	Government Sales
M100	01	S2	Non-Government Sales
M100	02	S1	Government Sales
M100	02	S5	Retail Sales

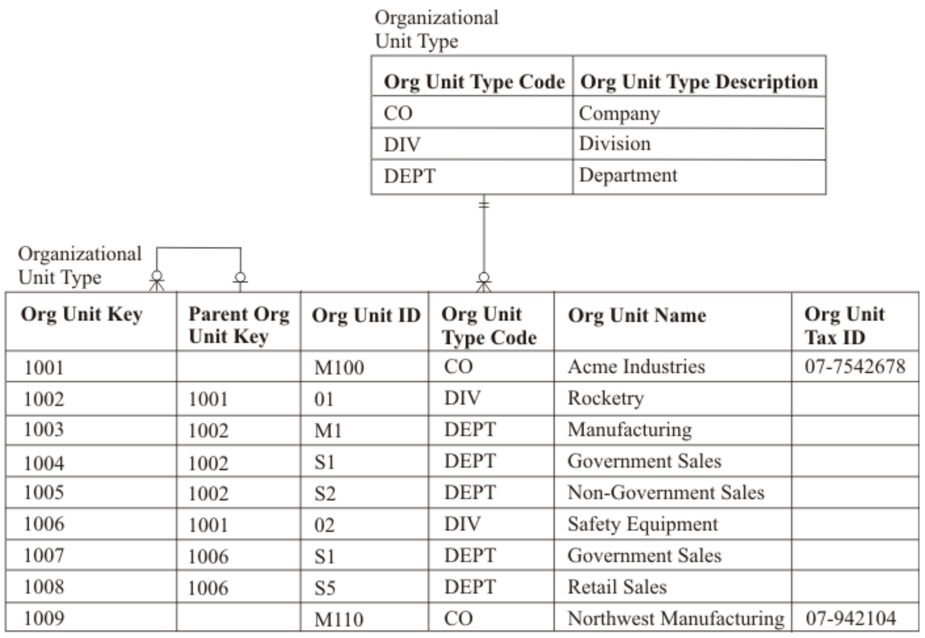
This is a very specialized hierarchical structure. The model is constructed using familiar one-to-many relationships, which yields a very straightforward representation. In fact, I know of several commercial payroll applications that use this same basic structure. However, the structure is quite rigid, leading to the following issues:

- If we need another layer, for example, we must add an entity and a new relationship to the model, which, of course, cascades all the way through to adding a table to the physical database.

- If some companies in the database have departments but no divisions, we will have to add a dummy division in their hierarchy to link their departments to the company layer.
- Even for companies with three-layer hierarchies, the structure can be confusing to use if they use different names for the layers. For example, a company that had departments at the second layer and workgroups under departments does not fit the structure well.

The following diagram shows a generalized structure that can accommodate an organizational hierarchy of any number of layers. The top of the diagram shows a conceptual model with attributes, and the physical representation at the bottom of the diagram shows the same data that appeared in previous diagram.




Organizational Unit Type
Organizational Unit Type

Org Unit Type Code	Org Unit Type Description
CO	Company
DIV	Division
DEPT	Department

Org Unit Key	Parent Org Unit Key	Org Unit ID	Org Unit Type Code	Org Unit Name	Org Unit Tax ID
1001		M100	CO	Acme Industries	07-7542678
1002	1001	01	DIV	Rocketry	
1003	1002	M1	DEPT	Manufacturing	
1004	1002	S1	DEPT	Government Sales	
1005	1002	S2	DEPT	Non-Government Sales	
1006	1001	02	DIV	Safety Equipment	
1007	1006	S1	DEPT	Government Sales	
1008	1006	S5	DEPT	Retail Sales	
1009		M110	CO	Northwest Manufacturing	07-942104

The Organizational Unit Type entity contains all the valid unit types such as Company, Division, and Department as used in previously shown specialized hierarchy. New types, which can be used to form new layers in the hierarchy, can be added by simply adding a new occurrence of the entity (in the database, this amounts to inserting a new row into a table).

The Organizational Unit entity contains all the organization units, regardless of their position in the hierarchy. The recursive one-to-many (Parent Org Unit Key is a foreign key for Org Unit Key) links the occurrences of Organizational Unit together into the desired hierarchy. You can see how this works by comparing the physical representation at the bottom of the diagram that shows the specialized hierarchy with the one at the bottom of generalized hierarchy shown in the preceding diagram.

The generalized version has the distinct advantage of flexibility—it can handle any number of layers and any type of organizational unit. Moreover, the layers do not have to be uniform. For example, some departments could report directly to the company, while others report to the company through a division. On the downside, it is more difficult for concrete thinkers to understand, and we are forced into using a generic identifier (Org Unit Key) instead of more familiar attributes such as Company ID, Division ID, and Department ID. Most experienced modelers immediately opt for the generalized structure, but if the organizational structure is expected to remain stable, a lot can be said for the simplicity of the specialized structure. The best guideline I can offer is to avoid generalizing for the sake of generalizing, but don’t hesitate to generalize when there is a reasonable possibility that flexibility will be needed to handle future changes in the structure.